

Ian Stewart og Robin Jones

Maskinkode med

COMMODORE 64



BORGEN

Maskinkode med COMMODORE 64

Ian Stewart og Robin Jones

Maskinkode med COMMODORE 64

BORGEN

Originalens titel:

Commodore 64

Machine Code

© Ian Stewart and Robin Jones, 1984

All rights reserved

Published by Shiva Publishing Ltd., 4 Church Lane, Nantwich
Cheshire CW5 5RQ England

Dansk udgave © 1985 Borgens Forlag

Oversat af Per Olesen

Omslag: Lars Beierholm

Trykt hos Narayana Press

ISBN 87-418-7396-3

2. oplag 1986

INDHOLD

Indledning	7
1. En lille appetitvækker	10
2. Tal i maskinkode	13
3. Positiv og negativ	20
4. Lageropbygning	24
5. 6510-mikroprocessoren	30
6. Et maskinkodeprogram	35
7. Indskrivning og udførelse af maskinkode	41
8. Assemblersprog	57
9. Adresseringsmåder	70
10. Flag	76
11. Forgrening og hop	83
12. Løkker	94
13. Indeksering	100
14. Indirekte adressering	108
15. Stak og underrutiner	117
16. Skærm- og farvekontrol	126
17. Tastaturkontrol	138
18. Logik	144
19. Sprites	149
20. Højopløsningsgrafik	169
21. MINIASS – en hjælp til gør-det-selv-assemblering	182
22. Nogle løse ender	195

Tillæg

1. Omregning mellem hex og decimal	200
2. Mnemonics	201
3. Oversigt over adresseringsmåder og mnemonic-formater	204

4. Opkoder for 6510 206
5. Operationernes indvirkning på flagene 208
6. Opkoderne i nummerorden til brug for
disassemblering 210
7. Spriteregistrene skematisk 212
8. Tastaturscanningskoder 213

Register 214

INDLEDNING

Commodore 64 er blevet en af de mest populære hjemme-computere i Europa og USA. Det er en alsidig og interessant maskine. Formålet med denne bog er at vise dig hvordan man kan udnytte dens muligheder endnu bedre ved at lære grundlaget for *maskinkodeprogrammering*. Vil du gerne på et øjeblik kunne fylde skærmen med et netværk af symboler? Eller flytte rundt med sprites hurtigt nok til at man kan spille et ordentligt spil? Eller tælle hvor mange gange REM-koden findes i et program? Så er det maskinkode du skal bruge. Maskinsprog stiller mange flere krav til programmøren end BASIC gør, men til gengæld udvider det antallet af opgaver som kan løses af computeren.

De fleste grundlæggende principper i denne bog gælder for alle computere der har en 6502- eller 6510-mikroprocessor. Men hele tiden har vi haft 64'erenes særlige egenskaber i tankerne, og skrevet teksten ud fra den antagelse at du sidder foran et varmt Commodore 64-tastatur mens du læser. Resultatet er en lempelig, men grundig introduktion til programmering i maskinkode og assembler, som ikke forudsætter nogen kundskaber ud over et beskedent kendskab til BASIC.

Vi begynder med at omtale hvordan tal fremstilles i maskinkode (hexadecimaltal, relative og absolutte binærtal, positive og negative tal), og hvordan – og hvor – koden lagres i hukommelsen. Dernæst kaster vi et blik på 6510- (og 6502-) processoren, din 64'ers hjerne, og dens interne opbygning, set fra programmørens synspunkt. Den har et antal særlige områder i hukommelsen, kaldet registre, og vi fortæller om hvad de gør. Så analyserer vi et mindre maskinkodepro-

gram udførligt, for at vise hvordan det adskiller sig fra BASIC.

Nogle af vanskelighederne med maskinkodeprogrammering kan undgås ved at lade computeren gøre arbejdet. Vi udvikler et BASIC-program (LOADER) som hjælper dig til at skrive, editere, indlæse og udføre maskinkode, samt at lagre programmer på bånd eller diskette og læse dem ind i hukommelsen igen. Dette program skal selv opbevares på bånd eller diskette, klar til brug i senere kapitler.

Når vores BASIC-toolkit er færdigt, er vi parat til at præsentere de vigtigste maskinkodeinstruktioner og nogle af de væsentlige teknikker: aritmetik, forgrening, løkker, flag, stak, underrutiner, logiske operationer. Det er den 'teoretiske' del, og den omfatter i alt væsentligt alle 6510-instruktioner.

I de afsluttende kapitler udvikler vi maskinkodeprogrammer som udnytter nogle af 64'erenes særlige egenskaber: sprites, farver, tastaturkontrol af bevægelig grafik, lav- og højopløsningsgrafik. Hovedvægten er lagt på enkle maskinkodeprogrammer, som er til at *forstå* og som kan benyttes som byggesten i mere komplicerede programmer. Vi vil gerne lære dig at *skrive* maskinkode, ikke bare at kopiere den.

Det er værd at lægge mærke til programmet MINIASS, som 'låner' Commodorens BASIC-editor og på snedig vis udnytter dens evner til at editere maskinkode i stedet (og sparer os for en masse besvær med at skrive en ordentlig editor). Maskinkoden læses så automatisk ind i hukommelsen fra BASIC-programområdet og er klar til udførelse.

For at runde denne snak af: vi har forsynet bogen med en hel række tillæg, som vil vise sig uvurderlige når der skal skrives maskinkode: skemaer over hex-/decimal-omregning, mnemonics, opkoder, adresseringsmåder, spriteregistre, flagenes funktioner og tastaturscanningskoder.

Denne bog giver en letforståelig, men grundig introduk-

tion til 6510- og 6502-maskinkode i almindelighed og til Commodore 64 i særdeleshed. Maskinkode er en udfordring der lønner sig. Prøv det!

Maskinkode kan bruges til at gøre ting med som simpelt hen ikke er mulige i BASIC. Dette kapitel skal være:

1 · EN LILLE APPETITVÆKKER

Du ville ikke have købt denne bog (eller bladret den igennem i boghandelen) hvis du ikke havde hørt at Commodore 64 kan udrette forbløffende ting meget hurtigt i noget der hedder *maskinkode*. Sandt nok – men problemet med maskinkode er at den – modsat BASIC – ikke tænker for dig. Du skal bruge meget mere tankevirksomhed på små fine detaljer, og holde øje med præcis hvor i hukommelsen koden ligger. Maskinkode er så afgjort *ikke* 'brugervenlig'; i begyndelsen ligner det hele mest ægyptiske hieroglyffer og er lige så umiddelbart tiltalende og forståeligt som en telefonbog på urdu.

Helt så slemt er det nu ikke, og med øvelsen får du hurtigt fornemmelse for det. Men du bliver helt sikkert nødt til at lægge en masse arbejde i det, før du virkelig får valuta for pengene. Og for at overbevise dig om at det nu også kan betale sig, vil jeg vise dig en maskinkoderutine som på et øjeblik kan ændre farve på karaktererne på skærmen. Den er lagt ind i et BASIC-program, og det kunne gøres endnu hurtigere hvis resten af programmet også blev skrevet i maskinkode. Det ville være noget nær umuligt at overtale BASIC til at løse den samme opgave med den kvarte hastighed (skønt jeg ikke vil sige det er helt umuligt, for folk er meget opfindsomme).

Prøv ikke på at forstå hvordan det alt sammen virker; det kommer senere. Skriv det bare af og RUN. Du vil bemærke et par kommandoer som du sikkert ikke bruger så tit, nemlig:

SYS

som siger til computeren at den skal udføre maskinkoden, og (nok bedre kendt)

PEEK

POKE

som laver numre med hukommelsen.

Så tager vi fat:

```
10 DATA 0,162,0,173,0,192,157,0,4
20 DATA 232,224,0,240,3,76,6,192,96
30 FOR T=0 TO 17
40 READ X
50 POKE 49152+T,X
60 NEXT
100 K=4
110 POKE 49152,INT(256*RND(0))
120 POKE 49160,K
130 SYS(49153)
140 K=K+1
150 IF K=8 THEN K=216
160 IF K=220 THEN K=4
170 GOTO 110
```

Undersøg nu *omhyggeligt* at du har skrevet det nøjagtig som det står – maskinkode kan spille én nogle grimme puds hvis der er fejl i. Tilfreds? Godt, RUN så programmet. Afbryd det hvis det bliver for irriterende for øjnene!

Som en variation kan du ændre linje 110 til:

```
110 GET A$:IF A$="" THEN 110
115 POKE 49152,ASC(A$)
```

Når du har startet programmet, prøv så at trykke på forskellige taster på tastaturet og se hvordan computeren reagerer. Det er ret hurtigt, hvad?

HVAD ER DET DER SKER?

Den måde det virker på, er at computeren bliver instrueret om at manipulere med to områder i hukommelsen: skærmhukommelsen (som rummer tegndata) og farvehukommelsen (som rummer farvedata). Maskinkoden i eksemplet ligger i DATA-linjerne 10 og 20, og i linje 30-60 læses de ind i et passende hukommelsesområde. Linje 130 får computeren til at udføre maskinkoderutinen, som begynder i det hukommelsesområde.

Nå, men jeg håber det har givet dig en idé om hvad en ganske enkel stump maskinkode (bare 18 bytes hukommelse og, som vi senere skal se, kun 8 maskinkodekommandoer) kan udrette. Små maskinkoderutiner kan betyde en væsentlig forbedring af et BASIC-programs muligheder.

Hvordan skal tal fremstilles i et maskinkodeprogram?

2 · TAL I MASKINKODE

Normalt tænker vi på tal i et system af tiere. Når jeg skriver tallet 3814 forstår vi alle at det betyder:

$$3 \times 1000 + 8 \times 100 + 1 \times 10 + 4 \times 1$$

og vi ser at hver gang vi rykker en position til venstre tæller cifret 10 gange så meget. Vi siger at tallet er på tier-basis.

Fordi vi har gjort det lige så længe vi kan huske, kan det godt være svært at indse at der er andre, lige så fornuftige måder at gøre det samme på. De folk der skabte de første computere indså det åbenbart ikke; de brugte tal udtrykt i basis 10 i deres maskiner, og rendte sig nogle grimme staver i livet. Det skyldtes for det meste at elektroniske forstærkere ikke opfører sig på samme måde ved alle signaler man tilfører dem. For eksempel kan en forstærker, som skal udsende et signal dobbelt så kraftigt som det den modtager, udmærket gøre det så længe den modtager 1, 2, 3 eller 4 signalenheder. Men så begynder den at 'flade ud', sådan at et signal på 5 kun giver en effekt på 9,6 og 6 kun frembringer 10,8; og måske kan man dårlig konstatere forskellen mellem effekten af signaler på 8 og 9.

Prøv at lægge et musikbånd på din billige kassettebåndoptager og skru op for lydstyrken. Kan du høre forvrængningen på de kraftige steder? Det er den samme effekt.

Computerpionererne hørte ikke nogen forvrængning; de mente bare at maskinerne somme tider ikke kunne skelne mellem forskellige cifre, og det var jo ikke så heldigt for en

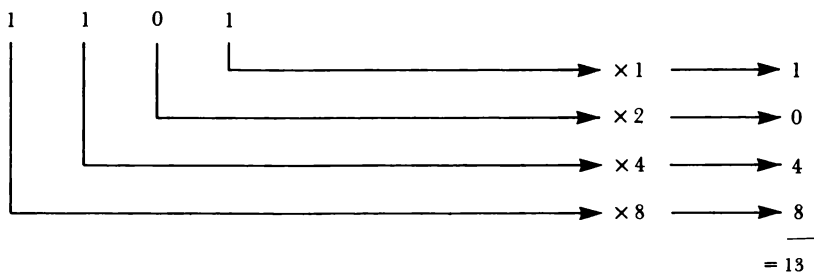
computer. De blev nødt til at tænke om igen og finde på en måde at notere tal på, som tog hensyn til hvad det elektroniske isenkram bedst kunne klare.

Den simpleste måde at behandle et elektrisk signal på er at tænde og slukke for det. På den måde kan man på en tilfredsstillende måde fremstille tallene 0 (slukket) og 1 (tændt). Så har forvrængningen ingen betydning længere. Det er helt klart om et signal er der eller om det ikke er der, uanset hvor forvrænget det er. Men kan man da udtænke et talsystem der kun bruger nuller og ettaller?

Ja. Med 10 som basis er 9 det størst mulige ciffer. Læg 1 til 9, så har du 10 – du har regnet med *mente*. Man kan skrive et hvilket som helst tal i en hvilken som helst basis, og det størst tænkelige ciffer vil altid være én mindre end basistallet. Hvis basis er 2 er det største ciffer 1, og tal i basis 2 (også kaldet *totalssystemet* eller *binærsystemet*) indeholder altså kun 0'er og 1'ere.

Hvad så med positionsværdierne? I titalssystemet fandt vi dem ved at begynde med 1 (til højre) og gange med 10 hver gang vi flyttede en position til venstre. Ved et binærtal begynder vi stadig med 1, men ganger med 2 hver gang vi går til venstre.

Således kan for eksempel binærtallet 1101 omregnes til basis 10 på denne måde:



At omregne den anden vej er også let nok. Tag for eksempel 25. Når vi noterer positionsværdierne for binærtal:

32 16 8 4 2 1

og går frem fra venstre, er det klart at vi først skal bruge 16. Så er der 9 tilbage, som består af en 8'er og en 1'er, så 25 hedder altså

0 1 1 0 0 1

HEXADECIMALKODE

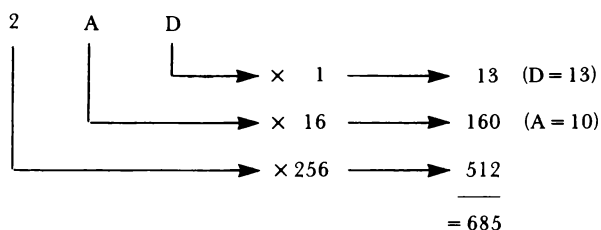
Det går fint med forholdsvis små værdier, men ved store bliver det noget rodet. Der findes en del teknikker til hurtig omregning, men jeg vil koncentrere mig om en fremgangsmåde der gør brug af *hexadecimalkode*, fordi det vil komme os til nytte senere.

Et hextal (der er aldrig nogen der siger 'hexadecimal', undtagen mig lige nu) er et tal med 16 som basis. Derfor finder man positionsværdierne ved hele tiden at gange med 16. De første fem er:

65536 4096 256 16 1

"Stop en halv!", hører jeg alle vegne fra. "Det er nogen grimme tal; det største ciffer i basis 16 må hedde 15. Nu bliver det for indviklet."

Bær over med mig. Problemet med cifre større end 9 vil vi klare ved at udstyre bogstaverne fra A til F med værdierne 10-15. Tallet 2AD i hex kan altså omregnes til titalssystemet sådan:



Nu til finessen ved at bruge hex. Eftersom 16 er en af positionsværdierne i binærsystemet (den femte), medfører det at ethvert ciffer i et hextal kan erstattes af de fire binærcifre som har den samme værdi. (Læg i øvrigt mærke til ordet 'bit' – det er en sammentrækning af den engelske betegnelse for binærciffer: 'binary digit'). Tabel 1 viser omregningsmulighederne:

Tabel 1

Decimal	Hex	Binær
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

En mere omfattende tabel findes i tillæg 1.

Lad os nu prøve at omregne 9041 til hex. Først uddrager vi to 4096'ere, så nogle 256'ere osv., sådan her:

$$\begin{array}{r} 9041 \\ 2 \times 4096 = \underline{8192} - \\ 849 \\ 3 \times 256 = \underline{768} - \\ 81 \\ 5 \times 16 = \underline{80} - \\ 1 \\ 1 \times 1 = \underline{1} - \\ 0 \end{array}$$

Tallet udtrykt i hex er altså 2351.

Nu skriver vi bare af efter cifferkoderne i tabellen:

2	3	5	1
0010	0011	0101	0001

og det er så hvad der svarer til 9041 i binærsystemet. De fire blokke skal bare sættes sammen til 0010001101010001.

Omregningen mellem hex og binær er så let at vi, mere som en regel end som en undtagelse, lader tallene stå i hex, selv om vi i sidste ende skal bruge dem i binær. Det er nu engang meget let at skrive fejl i en lang stribe nuller og et-taller.

OMREGNING PR. COMPUTER

Her er der et program der omsætter fra decimal til hex. Det dividerer hele tiden tallet med 16, og holder samtidig øje med resten. Det regner altså cifrene ud i omvendt rækkefølge af hvad jeg gjorde ovenfor.

```
20 LET H$=""
30 INPUT "DECIMALTAL";DT
40 N=INT(DT/16)
50 M=DT-16*N
60 IF M>9 THEN M=M+7
70 H$=CHR$(M+48)+H$
80 DT=N
90 IF DT>0 THEN 40
100 PRINT "HEXVAERDI ER: ";H$
```

Prøv programmet og omregn forskellige decimaltal til hex. (Det skal være hele positive tal: 0, 1, 2 osv.).

Dette program kan regne om den anden vej (hex til decimal):

```
110 LET DT=0
120 INPUT "HEXTAL";H$
130 FOR T=1 TO LEN(H$)
140 D$=MID$(H$,T,1)
150 A=ASC(D$)
160 A=A-48
170 IF A>9 THEN A=A-7
180 DT=16*DT+A
190 NEXT
200 PRINT "DECIMALVAERDI ER: ";DT
```

Vi kan knytte disse to rutiner sammen med en lille menu:

```
2 PRINT "DEC/HEX-OMREGNER"  
3 PRINT "1) DEC -> HEX"  
4 PRINT "2) HEX -> DEC"  
5 PRINT "3) SLUT"  
6 PRINT "VAELG 1, 2 ELLER 3"  
7 INPUT SVAR  
8 IF SVAR=1 THEN GOSUB 20  
9 IF SVAR=2 THEN GOSUB 110  
10 IF SVAR=3 THEN STOP  
15 GOTO 6
```

Og så skal der selvfølgelig være RETURN i linje 105 og 210.

For at kunne regne med negative tal bruger maskinen et smart lille kneb.

3 · POSITIV OG NEGATIV

Efter at vi har set lidt til hvordan binærtal udregnes, vil vi nu kigge på hvordan de håndteres inden i maskinen.

Almindeligvis opbevares et tal i et bestemt antal bit, ofte 16 eller 24 eller 32, afhængig af maskinens udformning. Dette antal bit kaldet maskinens *ordlængde*.

4-bits mønster	Decimalværdi
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Det er indlysende hvorfor der i praksis vælges større ordlængder; en maskine der kun kan fremstille tal mellem 0 og 15 er sandsynligvis helt utilstrækkelig. Men der er to andre problemer: denne notation kan ikke fremstille brøktal (7,14 for eksempel), og den kan ikke gengive negative tal.

Brøkproblemet vil vi se bort fra, for de fleste maskinkode-rutiner bruger kun heltal. Spørgsmålet om hvordan negati-ve tal virker er vigtigere lige nu.

Teknikken er enkel: hvis du har et positivt tal noteret som binærtal og vil finde det tilsvarende negative tal, gør du to ting:

1. Skift alle nuller ud med ettaller og alle ettaller med nuller (bittene 'vippes rundt').
2. Læg 1 til resultatet.

Lad os sige du vil finde -3 :

3 = 0011 i et 4-bits ord

Når bittene vippest rundt får vi

Læg så 1 til:

1100

+1

1101

1101 repræsenterer altså -3 . Det kaldes *tokomplementet* til 0011.

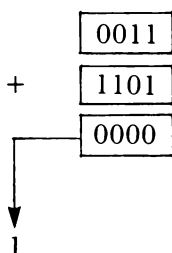
Jeg forklarer ikke nøjagtig hvordan det virker, men du kan i hvert enkelt tilfælde selv overbevise dig om at det gør. Lad os tage et eksempel:

Hvis vi lægger 3 til -3 (eller 5 til -5 eller alt mulig andet til minus sig selv) skal vi få nul. For eksempel:

$$\begin{array}{r} 111 \\ 0011 \quad (=3) \\ + 1101 \quad (= -3) \\ \hline = 10000 \end{array} \quad \text{(Husk at } 1 + 1 \text{ er lig } 0 \text{ og } 1 \text{ i mente i binærsystemet!)}$$

Så får vi altså *ikke* 0000 alligevel; men de fire mindst betydende bit *giver* nul, og når vi arbejder med et 4-bits ord vil den mest betydende bit bare falde ud over kanten. (For at tage et lignende eksempel fra dagligdagen: tænk på kilometertælleren i en bil. Hvis den har 4 cifre og viser 9999, og du så kører en kilometer til, så viser den 0000 og ettallet længst til venstre er ‘faldet af’.)

Vi kan med andre ord betragte det sådan:



Dette virker altid, under forudsætning af at antallet af bit er det samme overalt. Husk altid at sætte så mange nuller foran et tal at det får standardordlængden, *før* du finder to-komplementet.

Lad os skrive tabellen over 4-bits værdier om, sådan at den også indeholder negative værdier (se næste side).

Straks ser vi at der er et problem: hvert bitmønster optræder to gange, sådan at for eksempel 1001 kan betyde 9 eller -7. Vi bliver altså nødt til at indskrænke mængden af værdier yderligere. Jeg har tegnet en stiplede linje rundt om det område som vi reelt vil vælge at repræsentere. Ser du på den mest betydende (venstre) bit i hvert mønster, vil du kunne se at det er ‘0’ hvis tallet er positivt og ‘1’ hvis det er nega-

Decimal	Binær	Tokomplement	Decimal
0	0000	0000	0
1	0001	1111	-1
2	0010	1110	-2
3	0011	1101	-3
4	0100	1100	-4
5	0101	1011	-5
6	0110	1010	-6
7	0111	1001	-7
8	1000	1000	-8
9	1001	0111	-9
10	1010	0110	-10
11	1011	0101	-11
12	1100	0100	-12
13	1101	0011	-13
14	1110	0010	-14
15	1111	0001	-15

ti. Dette synes at være en oplagt måde at skelne dem fra hinanden på.

De tal der kan fremstilles med et 4-bits ord spænder altså over værdierne -8 til +7. Med 5 bit ville det være -16 til 15. For 6 bit ville det være -32 til +31 osv.

Et 16-bits ord (som er af betydning når vi taler 64'er) omfatter området -32768 til +32767. En tabel over notationen af tokomplement for 8-bits ord findes i tillæg 1.

For at programmere i maskinkode må du vide hvor i computeren information opbevares og i hvilken form.

4 · LAGEROPBYGNING

Som du sikkert ved har computeren to slags hukommelse (lager, hukommelse og memory er tre ord for det samme):

1. ROM (Read Only Memory) som indeholder fast information, der kan benyttes men ikke ændres af programmøren.
2. RAM (Random Access Memory) som kan ændres efter ønske.

Både ROM og RAM er opbygget på en måde som for programmøren ser ud som én lang liste over *lagerceller*. Hver celle kan lagre én *byte* information. En byte er et ord bestående af otte bit, fx 10011100; der er 256 mulige bytes, hvis decimalværdier går fra 0 til 255. En byte kan også fremstilles af et tocifret hexadecimaltal mellem 00 og FF.

Hver lagercelle har et nummer, en *adresse*, som den kan identificeres på. På 64'eren går de mulige adresser fra 0 til 65535 decimal. Hver adresse kan skrives som et fircifret hexadecimaltal mellem 0000 og FFFF. Det betyder at man kan fremstille en adresse ved hjælp af to bytes' (16 bits) information. Bemærk at $65536 = 256 \times 256$. En *kilobyte* er 1024 bytes' hukommelse, og 65535 bytes' hukommelse er det samme som 64 kilobytes (64 K) – det er deraf 64'eren har sit navn.

(I virkeligheden er det ikke helt rigtigt, for 64'eren har yderligere nogle lagerområder til særlige formål. Andre hu-

kommelses-‘banker’ kan kobles til eller fra når der er brug for det. Se *Programmer's Reference Guide** s. 260. Jeg vil springe denne mulighed over for ikke at gøre historien for lang.)

Vi kan altså, uden at gå ned i detaljerne, afbilde hukommelsen sådan:

Adresse i decimal		Adresse i hex
0		0000
1		0001
2		0002
3		0003
.		
.		
.		.
65533		FFFD
65534		FFFE
65535		FFFF

Hele diagrammet i denne målestok ville være omkring 400 meter langt!

PEEK OG POKE

Fra BASIC kan man få direkte adgang til en lageradresse. Med kommandoen

PEEK

* *Commodore 64 Programmer's Reference Guide*

– fås hos din Commodore-forhandler

kan man se hvad der ligger i den (den virker både i ROM og RAM), og med

```
POKE
```

kan man ændre indholdet i den (kun i RAM).

For at finde indholdet af adresse AD skal du bruge

```
PEEK(AD)
```

hvor AD skal være en decimalværdi. Prøv for eksempel dette program:

```
100 FOR AD=900 TO 920
110 PRINT AD, PEEK(AD)
120 NEXT
```

Start programmet, så får du en liste over (decimal)-indholdet i de lagerceller hvis adresser går fra 900 til 920 (decimal).

Kommandoen POKE anvendes i denne form:

```
POKE AD, TAL
```

hvor AD er adressen, TAL tallet der skal lægges ind i den (0 – 255 i decimal). Prøv for eksempel at tilføje denne rutine til de tre programlinjer ovenfor:

```
10 FOR AD=900 TO 910
20 POKE AD, 77
30 NEXT
```

Kør hele molevitten. Du vil opdage at indholdet i adresserne 900-910 nu er blevet 77 (det samme som 4D i hex).

Der er nogle områder i RAM hvor det ser ud til at POKE ikke har den forventede virkning. Det skyldes BASIC-opera-

tivsystemet, som bruger visse dele af hukommelsen og smadrer dine POKE-værdier. Adresserne 900-920 ovenfor ligger faktisk i et område der kendes som *kassettebufferen*, som ikke bliver smadret medmindre du LOADER eller SAVER programmer. Prøv at LOADE et program og derpå PEEKE adresserne 900-910. Viser de stadig 77?

Det er en fælde som vi må passe på, når vi siden hen skal til at lagre maskinkode. Ikke fordi det er vanskeligt at finde et sikkert sted at lægge den. Men det er vigtigt at gøre det.

Maskinkodeprogrammer er meget lidt fleksible hvad angår måden adresserne angives på. Adresser er altid fircifrede hexnumre, såsom

A1C7 FFFC 55D0 0007

og nullerne foran tallet, som i det sidste eksempel, *skal med*.

SIDER

Hvert afsnit på 256 bytes i hukommelsen kendes under navnet en *side* eller *page*. Der er altså i alt 256 sider. De to første cifre i hexadressen angiver *sidenummeret* (*page number*). For eksempel ligger adresserne ovenfor på siderne

A1 FF 55 00

i den rækkefølge. *Side nul* (*page zero*) er specielt for maskinkode og den behandles temmelig forskelligt fra alle andre sider.

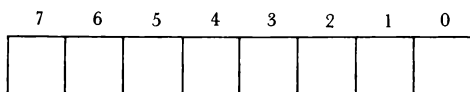
OVERSIGT OVER LAGERET

Du bliver nødt til at kunne finde rundt i 64'ers hukommelse, hvis du vil have nogen indflydelse på hvordan bæstet opfører sig. På en computer i standardkonfigurationen er de vigtigste områder i hukommelsen disse:

Decimal	Hex	Bruges til
0 – 827	0000 – 033B	Operativsystem
828 – 1019	033C – 03FB	Kassettebuffer
1024 – 2023	0400 – 07E7	Skærmhukommelse
2040 – 2047	07F8 – 07FF	Spritedata-pointere
2048 – 40959	0800 – 9FFF	BASIC-område
40960 – 49151	A000 – BFFF	BASIC ROM <i>eller</i> 8K RAM
49152 – 53247	C000 – CFFF	4K RAM
53248 – 54271	D000 – D3FF	VIC-chip (sprites, tegngenerator)
54272 – 55295	D400 – D7FF	SID-chip (lyd)
55296 – 56319	D800 – DBFF	Farvehukommelse
56320 – 57343	DC00 – DFFF	Input/output m.v.
57344 – 65535	E000 – FFFF	KERNAL ROM <i>eller</i> 18K RAM

BIT-NUMMERERING

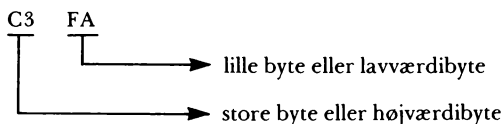
Den traditionelle måde at nummerere de enkelte bit i en byte på er denne:



Bit 0 bidrager således med 1 til værdien, bit 1 med 2, bit 2 med 4; generelt udtrykt: bit N bidrager med $2^{\uparrow N}$. De mest

betydende bit (dem længst til venstre) har de højeste numre og den største andel i byte'ns værdi (ganske som cifrene i decimaltal).

På tilsvarende måde udgør de to cifre til venstre i et hex-tal en *højværdibyte* (*high byte*), og de to til højre udgør en *lavværdibyte* (*low byte*). I denne bog vil vi bruge de nemmere betegnelser: *store byte* og *lille byte*. Et eksempel:



*Må jeg præsentere . . .
hjernen bag virksomheden:*

5 · 6510-MIKROPROCESSOREN

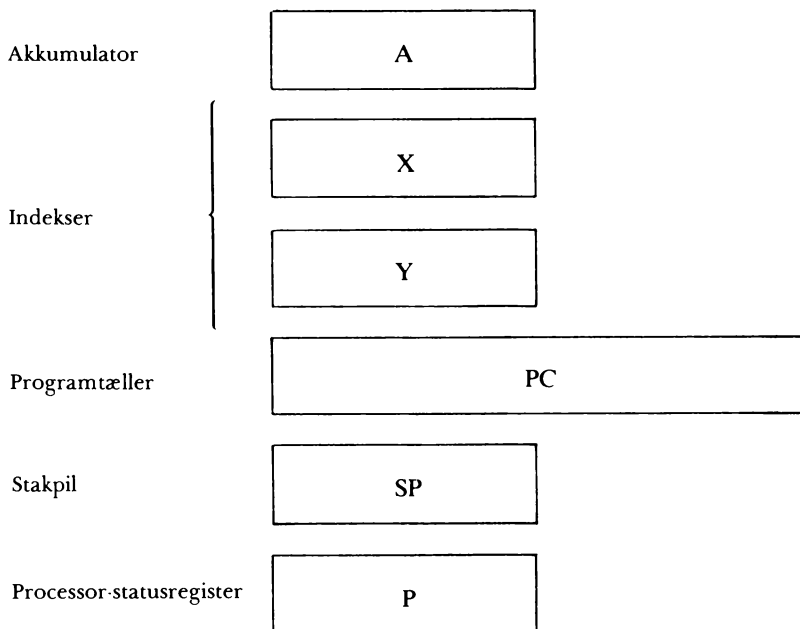
I hjertet (eller hjernen) på din 64'er sidder der et imponerende stykke teknik (om end en smule gammeldags efter dagens standard): mikroprocessorchippet 6510. Den er din computers *centralenhed* eller CPU (Central Processing Unit), og den indeholder alle de kredsløb der er nødvendige for at udføre logiske og aritmetiske operationer og kontrollere hvordan alt det andet virker. Den er en modificeret version af den berømte MOS-chip 6502, og for så vidt angår maskinkodeprogrammer er de to identiske. (Jeg nævner det fordi de fleste tilgængelige bøger handler om 6502; dem kan du roligt købe, i sikker forvisning om at alt hvad der står i dem lige så vel gælder for 6510).

Af en mikroprocessor at være er 6510 nogenlunde enkel; men der er en del mindre komplikationer og biproblemer, som jeg ærligt talt helst er fri for at komme ind på. En bog fuld af hvis'er og men'er og måske'er er som et forhindringsløb. Lad mig derfor allerede nu advare dig om at jeg ikke vil fortælle *hele* sandheden. Jeg vil hellere skøjte hen over spidsfindighederne end forskanse mig bag forvirrende men kvalificerede kommentarer, som ikke har betydning for andre end eksperten.

I det konkrete tilfælde: det har ikke nogen betydning for os at kende 6510-processorens eksakte fysiske opbygning; hvad vi derimod har brug for at vide, er hvordan man skal forestille sig den når man skriver et program. Lad os derfor kaste et blik på hovedtrækkene.

REGISTRENE

Inden i 6510-processoren er der et antal specialiserede hukommelsesområder, *registre*, som den bruger til at udføre instruktioner med. Man kan forestille sig at de er arrangeret sådan:



Hvert register indeholder én byte, undtagen PC-registret der indeholder to bytes. Om et øjeblik vil du se hvorfor. Til orientering er der her en hurtig gennemgang af hvad de allesammen laver. Jeg fortæller mere senere, når vi skal til at gøre brug af dem.

AKKUMULATOREN

Akkumulatoren er grundlaget for alle aritmetiske og logiske operationer. For eksempel skal man for at lagre en bestemt byte i hukommelsen (som med POKE i BASIC):

1. Indlæse den i akkumulatoren.
2. Lagre akkumulatorens indhold i hukommelsen.

Du vil synes at du bruger en masse tid på at skovle ting ind i akkumulatoren og hale dem ud igen. Hvis du vil lægge to tal sammen (eller trække det ene fra det andet) må du lægge det ene ind i akkumulatoren, så addere (eller subtrahere) det andet, og til sidst se efter i akkumulatoren hvad resultatet blev.

Eftersom akkumulatoren kun er 8 bit bred kan man kun lave aritmetik med tal op til 255. Vi skal senere også få at se hvordan man kommer ud over det.

Når man har programmeret i BASIC med dets ubegrænsede udvalg af variabler, tager det et stykke tid at vænne sig til den grusomme kendsgerning at der *kun er én akkumulator*. En god måde at forestille sig denne situation på er at tænke på en lommeregner med 8-ciffrers display. Hver gang du har foretaget en udregning, kommer resultatet ud på displayet. Hvad der end stod der før, så er det væk nu – *medmindre* du har været så forudseende at gemme det i memory først. Med akkumulatoren er det helt det samme.

INDEKSREGISTRENE

6510 har to *indeksregistre*, X og Y. De indeholder tal som kan bruges til at gennemløbe områder af hukommelsen trin for trin. De er nyttige ved lister og tabeller, og hvor der ellers er brug for at noget skal gøres ved en hel blok af hukommelsen. De kan også bruges når man i maskinkode vil flikke en løkke sammen der svarer til FOR/NEXT i BASIC.

PROGRAMTÆLLEREN

PC-registret (PC for Program Counter) bruger man kun indirekte, og normalt behøver man ikke spekulere på hvad den laver. Den giver besked til CPU'en om hvilken programinstruktion den skal udføre næste gang. Det er vigtigt, for man kan få programmet til at hoppe til en kommando ved at ændre den værdi der ligger i PC-registret. Det er hvad der svarer til BASIC's GOTO-kommando. I virkeligheden rummer PC-registret *adressen* på den lagercelle som indeholder koden for næste kommando. Da adresser jo er to-bytes tal, må PC-registret også være to bytes langt. Det er grunden! Flere oplysninger kan findes i kapitel 11.

STAKPILEN

Der er i 64'eren's hukommelse et særligt område kaldet *stakken* (se kapitel 15) som bruges til midlertidig datalagring under udregninger. SP-registret fortæller CPU'en hvor stakken slutter. Stakken er også af afgørende betydning ved brug af *underrutiner* i maskinkode.

PROCESSORSTATUSREGISTRET

P-registret indeholder information som kan bruges til at træffe beslutninger ud fra. Er et nummer positivt? Er det negativt? Eller 0? Har en aritmetisk operation givet én i mente? Hver gang en kommando udføres, opdateres P-registret automatisk. (Se kapitel 10 om *flag*).

Det var de nøgne kendsgerninger, men der er selvfølgelig mere at fortælle. (Det er der *altid* når man taler computer). For at vi kan vænne os smertefrit til maskinkode, vil vi først

kaste et blik på nogle enkle men oplysende eksempler. Så vil vi være klar til at snakke om hvordan processorens registre og kommandoer udnyttes effektivt.

Den bedste måde at forstå hvordan et maskinprogram skal skrives er at se hvad der sker når computeren arbejder sig gennem et enkelt eksempel.

6 · ET MASKINKODEPROGRAM

Formålet med dette kapitel er at vise hvilken form et maskinkodeprogram har når det lagres i hukommelsen, og hvad det er for et nydeligt benarbejde der foregår inden i CPU'en når den kører programmet. Jeg vil begynde med et af de simplest mulige programmer: en 8-bit-additionsrutine. Den tager to tal mellem 0 og 255 (decimal) og adderer dem. Det ville være meget nemt i BASIC:

```
10 INPUT M,N
20 L=M+N
```

I maskinkode da? Tja, vi får se!

I BASIC vænner vi os hurtigt til den tanke at en bestemt byte i hukommelsen kan have mere end én betydning. For eksempel kan det være et tal eller ASCII-koden for et tal eller instruktioner der kontrollerer en sprite. Betydningen afhænger ikke så meget af hvor det står som *hvad computeren har i sinde at gøre med det*. Og *det* er fastlagt af den person der har tegnet maskinens kredsløb.

Det samme er tilfældet i maskinkode. Indholdet i en bestemt lagercelle kan behandles som et positivt tal, et relativt tal mellem -128 og 127 eller som en instruktionskode. Hvis du skriver programmet korrekt vil computeren altid vide hvilken betydning du har ment. Men hvis du laver en fejl er der en oplagt risiko for at computeren vil blive forvirret. Og

når et maskinkodeprogram løber løbsk kan der somme tider komme de underligste ting ud af det.

PROGRAMMET

Først vil jeg vise hvordan programmet ser ud når det ligger i hukommelsen (se tabel 2 næste side). Jeg vil lægge det fra adresse 49152 og frem, svarende til C000 (hex) og frem. Denne ene gang vil jeg opgive hukommelsens indhold i både hex, binær og decimal. (Hex er hvad du må lære at tænke i for at bruge maskinkode, binær er hvad der faktisk er til stede i hardware't, og decimal er hvad du får at se med PEEK).

Jeg har her gjort en del ting for at hjælpe os til at se hvad der sker. For det første har jeg givet to af hukommelsens områder navne: 'data' og 'program'. Programmet skal bruge dataområdet til lagring af variabler. Jeg har også trukket vandrette streger for at inddеле programmet efter dets enkelte kommandoer; bemærk at nogle er tre bytes lange, nogle kun én byte (to-bytes kommandoer findes også, bare ikke i dette program).

Når programmet er lagt ind i computeren 'ved' den ikke noget om disse ting; den har bare en liste med bytes. Men når programmet kører kan den ud fra *sammenhængen* regne ud om den enkelte byte indeholder program eller data eller hvad.

HVAD SKER DER NÅR PROGRAMMET KØRER?

Lad os vente til næste kapitel med at lægge disse bytes ind i de rigtige adresser; det er langt fra venstrehåndsarbejde, men lad os i stedet se hvad CPU'en gør når den bliver instrueret om at udføre dette program. Selve programmet be-

Tabel 2

Adresse		Adressens indhold		
		Hex	Binær	Decimal
data	C000	07	00000111	7
	C001	05	00000101	5
	C002	00	00000000	0
program	C003	18	00011000	24
	C004	AD	10101101	173
	C005	00	00000000	0
	C006	C0	11000000	192
	C007	6D	01101101	109
	C008	01	00000001	1
	C009	C0	11000000	192
	C00A	8D	10001101	141
	C00B	02	00000010	2
	C00C	C0	11000000	192
	C00D	60	01100000	96

gynder på adresse C003, og programmøren giver bolden op ved at bede computeren læse denne adresse ind i program-tællerregistret.

Computeren 'ved' nu at der kommer en instruktion som den skal tyde. Den bruger adressen i PC-registret til at finde koden, som er 18. Chippens indbyggede kredsløb fortæller den at det betyder 'slet flaget for mente'. Det hentyder til P-registret og er en slags lille hovedrengøring, som er nødvendig for at sikre at al ting starter pænt og nydeligt, uden fodspor efter det tidligere program.

Computeren 'ved' også at kode 18 er en *1-bytes kode*; det vil sige at den for at finde næste kommando må forhøje PC med 1. PC har nu C004. Denne adresse indeholder koden AD, som betyder 'lad akkumulatoren med det tal som er lagret i den adresse som er angivet i de to næste bytes i programmet'. De to næste bytes er 00 og C0. Computeren sætter dem sammen i rækkefølgen C0, 00, og har så adressen C000. Den adresse ligger i dataområdet og indeholder 07. Derfor lægger computeren *tallet* 07 ind i akkumulatoren.

Dette har nu lagt beslag på *tre* bytes af programmet: AD,00,C0. AD er med andre ord en *3-bytes kode*. PC må flyttes 3 opad for at finde næste instruktion. Nu indeholder PC altså C007 (adressen på den første programbyte efter de netop udførte: AD,00,C0).

Nu er du nok så småt ved at fange ideen. Computeren tyder nu det der ligger i C007. Det er 6D, som betyder 'addér akkumulatorens indhold med det tal som ligger i den adresse som er angivet i de to næste bytes i programmet'. De to næste bytes er 01,C0; de henviser til C001, som indeholder 05. Derfor lægger CPU'en 05 til de 07 som allerede er i akkumulatoren og får resultatet 0C (i hex, er du med? $5 + 7 = 12$ i decimaltal, det samme som 0C i hex). Også det var en 3-bytes kode, og PC flytter altså op til C00A.

Her står koden 8D, som betyder 'gem akkumulatorens indhold i den adresse som er angivet i de to næste bytes i

programmet'. De to næste bytes, 02 og C0, henviser til adresse C002. Altså gemmer computeren tallet 0C i adresse C002. Der var igen en 3-bytes kode, så PC går op til C00D.

Når computeren nu tyder indholdet i C00D, nemlig 60, finder den ud af at den skal 'vende tilbage til BASIC'. Det gør den, og den afslutter dermed maskinkodekørslen. Det sidste var kun en 1-bytes kode, og PC flytter én op til C00E; men nu er vi tilbage i BASIC, som straks overtager kontrollen over alle registre.

HVAD SKER DER I PROGRAMMET?

Og hvad har det udrettet? Det tog indholdet i adresse C000, lagde det til indholdet af C001 og gemte svaret i C002. Det er en 8-bit-sammenlægger. Hvis vi ændrer indholdet af C000 til 1A (decimal 26) og C001 til 0E (decimal 14), så ville C002 bagefter indeholde summen af de to tal: 28 (decimal 40). Og så fremdeles.

Faktisk er det lidt det samme som BASIC-instruktionen $L = M + N$, hvor vi bare har valgt at bruge adresse C000 for variabelen M, C001 for N og C002 for L. Læg mærke til at det er dig der bestemmer hvor disse variabler skal lægges hen. BASIC plejer automatisk at finde ud af det for dig. I maskinkode er du henvist til dig selv.

OPKODER

De af kodens bytes som medfører en bestemt operation inden i CPU'en kaldes operationskoder eller *opkoder*. Programmet ovenfor kan opløses i opkoder på denne måde:

07	}	Data
05		
00		
18		Opkode for 'Slet flaget for mente'
AD 00 C0		Opkode for 'Lad akkumulator fra adresse C000'
6D 01 C0		Opkode for 'Addér indholdet af adresse C001'
8D 02 C0		Opkode for 'Gem resultat i adresse C002'
60		Opkode for 'Vend tilbage til BASIC'

6510 har 56 forskellige kommandoer, men de fleste kan anvendes på mange forskellige måder (kaldet *adresseringsmåder*; se kapitel 9). Der er 151 forskellige opkoder. Vi behandler alle de væsentlige i bogens løb. Nogle kræver 1 byte, nogle 2 og nogle 3. Men vi behøver *ikke* lære koderne udenad! De står alle sammen i tillæg 4.

Computeren kan bringes til at håndtere det meste af rutinearbejdet ved maskinkodeprogrammeringen . . .

7 · INDSKRIVNING OG UDFØRELSE AF MASKINKODE

Det er ikke vanskeligt at udføre et maskinkodeprogram. De fleste problemer opstår under indskrivningen (og under fejlfinding, som er et kapitel – eller en hel bog – for sig!). Ved at skrive passende BASIC-rutiner kan man spare sig for en hel del besvær. Hovedsigtet med dette kapitel er at udvikle rutiner af den art. Man kunne godt skrive dem meget udførligt, men jeg vil gerne holde listningerne nede på en rimelig længde, så vi kan koncentrere os om hovedemnet: selve maskinkoden.

HVOR LAGRES MASKINKODE?

I princippet kan man lægge sin maskinkode hvor som helst i RAM – men, som jeg tidligere har nævnt: i praksis vil BASIC smadre det hvis man lægger koden i et område som BASIC tilfældigvis bruger.

En tiltalende løsning – og den som jeg vil lægge mig fast på i denne bog – er at bruge det 4K lange RAM-område fra adresse C000 til CFFF (49152 – 53247 decimal). Dette område bruges ikke af BASIC og er et sikkert sted at lægge sin kode (du har nok bemærket at din 64'ers højt berømmede 64K RAM på mirakuløs vis bliver til '38911 BASIC bytes free' når du tænder for dyret; denne blok på 4K RAM er en af forklaringerne).

Et andet sted som folk tit bruger til ganske små maskinkoderutiner er kassettebufferen, 033C til 03FB (828-1019 decimal). Det kan være udmærket hvis man ikke skal bruge kassettebåndoptageren, men det kan ikke anbefales at programmere på den måde.

En anden metode – som du bliver *nødt* til at bruge hvis du (himlen forbyde det!) har mere end 4K maskinkode – er at ændre på de pointere som fastlægger grænserne mellem de forskellige områder af hukommelsen. For eksempel kan man flytte begyndelsen af BASIC-området længere ned, og på den måde skaffe sig ledig plads som BASIC ikke længere kan lægge sin klamme hånd på. Se *Programmer's Reference Guide* for nærmere detaljer. I denne bog vil jeg holde mig til området C000–CFFF. Dette område vil jeg omtale som *standardområdet*.

INDSKRIVNING MED DATALINJER

Der findes en meget enkel måde at indskrive maskinkode på, som jeg brugte i kapitel 1. Den har mange ulemper, som jeg straks vil komme ind på, men til små rutiner, som man allerede har gennemtestet og bare vil have liggende klar til brug, er det somme tider den nemmeste og hurtigste løsning. Princippet er at listen over bytes der skal indlæses bliver lagt i datalinjer, hvorfra de POKEs på plads i en programløkke. For eksempel kan vi indlæse vores 8-bit-sammenlægger på denne måde:

```
10 DATA 7,5,0,24,173,0,192,109,1,192,141,2,192,  
    96  
20 FOR T=0 TO 13  
30 READ X  
40 POKE 49152+T,X  
50 NEXT
```

Fordelene er ret indlysende. Til ulemperne hører:

1. At koden skal omsættes fra hex til decimal.
2. At maskinkoden optræder *to gange* – dels på sin endelige, 'indlæste' plads, dels i DATA-linjerne. Man spilder lagerplads.
3. At det er let at gøre fejl under indtastningen, hvis DATA-listen er bare nogenlunde lang.
4. At det er svært at læse og rigtig forstå DATA-linjer.

Nogle af disse problemer kunne dog undgås hvis man brugte en DATA-liste med hexkoder, behandlet som strenge af to karakterers længde, og tilføjede nogle programlinjer som kunne omsætte dem til decimaltal. Måske kan du bruge den idé til noget.

Men med en lille ekstra indsats kan vi udvikle et BASIC-program, som ikke blot tillader os at indlæse maskinkode, men også at liste den, ændre i den og minsandten også udføre den. Dét vil vi beskæftige os med.

Pas på: Vær sikker på at kassettebåndoptageren er tilsluttet (sluk først for computeren) før du går videre. Der forestår et længere tastearbejde, og du er nok ikke interesseret i at gøre det om! Det kan endda være du vil skrive alle de følgende programlinjer og SAVE dem på bånd *inden* du afprøver programmet.

EN HEXLOADER

Jeg går ind for det synspunkt, at i maskinkodeprogrammering skal man tage alle de faciliteter med som man kommer billigt til. Derfor udskriver programmet adresser og koder i både decimal og hex. Det giver dig tilbud om at vælge hvor

du vil lægge koden. Og det vil give dig mulighed for at have et dataområde forud for programområdet.

Jeg giver dig programmet i småbidder for at gøre det hele mere begribeligt. Først har vi en menu:

```
90 PRINT CHR$(147)
100 PRINT "HEXLOADER: VALGMULIGHEDER"
110 PRINT "L:LOAD  P:PRINT  E:EDIT  R:RUN
      S:STOP"
120 GOSUB 1300
130 IF Q$="L" THEN 200
140 IF Q$="P" THEN 800
150 IF Q$="E" THEN 1000
160 IF Q$="R" THEN 1200
170 IF Q$="S" THEN STOP
180 GOTO 110
```

Før det kan køre må vi have en lille input-rutine:

```
1300 GET Q$:IF Q$="" THEN 1300
1310 RETURN
```

Programmet har en del enkeltbogstavs-input; med denne metode undgår du at skulle trykke RETURN evig og altid.

Nu kommer LOAD-rutinen:

```
200 PRINT "LOAD DATA OG PROGRAM"
210 PRINT "BEGYNDELSSESADRESSE I DECIMAL
      (DEFAULT 49152)"
220 INPUT BA
230 IF BA=0 THEN BA=49152
240 PRINT BA:PRINT
250 INPUT "ANTAL DATABYTES";D
260 AD=BA
270 PRINT:PRINT "SKRIV KODE I HEX"
```

```

280 PRINT "TRYK S FOR STOP":PRINT
290 PRINT "HEXADR","DECADR","HEXKODE","DECKODE"
300 IF AD=BA AND D>0 THEN PRINT "*DATA*"
310 IF AD=BA+D THEN PRINT "*PROGRAM*"
320 GOSUB 500
330 PRINT HA$,AD,
340 GOSUB 1300:H$=Q$
350 PRINT H$;
360 IF H$="S" THEN 90
370 GOSUB 1300:L$=Q$
380 PRINT L$,
390 GOSUB 600
400 PRINT DK
410 POKE AD,DK
420 AD=AD+1
430 GOTO 300

```

Dette medfører et par rutiner til omregning mellem hex- og decimalkode, baseret på dem vi havde i kapitel 2. Den første er:

```

500 HA$="":AM=AD
510 FOR T=1 TO 4
520 N=INT(AM/16)
530 M=AM-16*N
540 IF M>9 THEN M=M+7
550 HA$=CHR$(48+M)+HA$
560 AM=N
570 NEXT
580 RETURN

```

Og her er den anden:

```

600 H=ASC(H$):H=H-48:IF H>9 THEN H=H-7
610 L=ASC(L$):L=L-48:IF L>9 THEN L=L-7

```

620 DK=16*H+L

630 RETURN

Der kommer mere, men nu vil vi kaste et blik på

HVORDAN LOADEREN BRUGES

Som eksempel tager jeg maskinkodeprogrammet fra kapitel 6 igen: 8-bit-sammenlæggeren. Du husker at den havde tre databytes og lå fra adresse C000 og frem – i standardområdet. Den komplette hexkode for programmet er:

```
07 05 00 18 AD 00 C0 6D 01 C0 8D 02 C0 60
```

og vi vil gerne have LOADERen til at lægge koden på plads.

Kør LOADER-programmet. Menuen kommer frem; tryk på L for at vælge LOAD. Programmet spørger efter begyndelsesadressen i decimal og oplyser at 'default' er vores foretrukne: 49152 (standardområdet i decimal). Hvis du taster 0 eller RETURN vil programmet automatisk an vise den som begyndelsesadresse for maskinkoden (hvis du vil have en anden startadresse skriver du den i stedet – i decimal).

Du bliver nu spurgt om antallet af databytes; det er 3, så det skriver du. Computeren beder dig nu indskrive koden i hex, og minder dig om at du ved at trykke S kan stoppe LOAD-indskrivningen.

Derefter skriver den fire spalteoverskrifter, som står for Hexadresse, Decimaladresse, Hexkode og Decimalkode. Den husker dig venligt på at du nu skal indskrive

DATA

Startadressen bliver vist både i hex og i decimal:

C000 49152

Så er det din tur, og du trykker 0 og 7, som er de to første hexcifre i maskinkoden. Skærmen viser nu

C000 49152 07 7
C001 49153

og så kan du skrive de næste to hexcifre ind: 05. Fortsæt med at skrive maskinkode indtil du når til de 60 til sidst (du får en meddelelse når *PROGRAM*-området begynder). Nederst på skærmen står der så

C00D 49165 60 96
C00E 49166

Tryk S, for nu er vi færdige. Programmet vender tilbage til menuen; med S igen kan programmet stoppes.

LOADER virker på samme måde ved alle andre rutiner. Først opgiver du den begyndelsesadressen (eller trykker RETURN for default-værdien), så antallet af databytes (0 hvis der ingen er), og derefter skriver du hexkoderne ind i rækkefølge, to cifre ad gangen, og slutter med et S når du er færdig. Computeren gør resten, og du får udskrifter på skærmen efterhånden.

PRINT-FUNKTIONEN

Da skærmen ruller mens du taster ind, vil du altid kun kunne se de ca. 20 sidste linjer. Hvis du har skrevet flere skærmfulde* må du kunne checke dem én ad gangen. Netop af den grund har LOADER en PRINT-funktion:

* Eller hedder det skærme fuld?

```

800 PRINT CHR$(147);"PROGRAMLISTNING"
810 AD=BA
820 PRINT "HEXADR","DECADR","HEXKODE","DECKODE"
830 FOR K=0 TO 19
840 DK=PEEK(AD)
850 GOSUB 700
860 GOSUB 500
870 PRINT HA$,AD,HK$,DK
880 AD=AD+1
890 NEXT
900 GOSUB 1300
910 IF QS="S" THEN 100
920 GOTO 820

```

Også her er der en kodeomregning:

```

700 H=INT(DK/16):L=DK-16*H
710 H=H+48:IF H>57 THEN H=H+7
720 L=L+48:IF L>57 THEN L=L+7
730 HK$=CHR$(H)+CHR$(L)
740 RETURN

```

For at afprøve det skal du trykke på P når menuen er kommet frem, så får du en skærmfuld af listningen. Tryk S for at stoppe eller en hvilken som helst anden tast for at få næste skærmfuld. (NB: Hvis du vælger P uden at have fastsat begyndelsesadressen BA, vil computeren sætte den til 0. En måde at komme uden om det på er at tilføje linjen

```
805 IF BA=0 THEN BA=49152
```

og altså give BA default-værdien.)

AT STARTE ET MASKINKODEPROGRAM

er nemt nok. BASIC-kommandoen

SYS

gør arbejdet for dig. For at starte maskinkodeprogrammet der begynder på adresse AD skal du bruge

SYS(AD)

Vores 8-bit-sammenlægger kan altså startes med kommandoen

SYS(49155)

hvor selve sammenlægningsprogrammet begynder (C003 i hex). Lad os én gang for alle tilføje en RUN-rutine til LOADER:

```
1200 PRINT:PRINT "KLAR"
1210 PRINT "TRYK PAA EN TAST (S FOR AT AFBRYDE)"
1220 GOSUB 1300
1230 IF QS="S" THEN 100
1240 SYS(BA+D)
1250 PRINT:PRINT "PROGRAM UDFOERT":PRINT
1260 GOTO 100
```

Tilføj disse linjer til LOADER – så er du køreklar. Men bemærk at du første gang må indskrive værdierne for begyndelsesadresse (BA = 49152) og antal databytes (D = 3) manuelt, ellers vil linje 1240 ikke blive udført korrekt. Det samme gælder hver gang du har foretaget tilføjelser eller ændringer til programmet. Start så med GOTO 90 (ikke RUN!). Nu kan du:

1. Vælg L for at indskrive koden for 8-bit-sammenlæggeren (hvis du ikke allerede har gjort det).
2. Vælg P for at checke at koden er rigtig.
3. Vælg R for at starte.

Computeren vil vente på at du trykker på en tast (og du kan vælge at trykke S og slippe for at programmet kører, hvis du pludselig er kommet i tanker om en frygtelig fejl). Tryk noget andet end S.

Lynhurtigt kommer meddelelsen

PROGRAM UDFOERT

og menuen.

Udmærket, men hvor er svaret henne?

Jo, du husker vel at vi gemte resultatet af sammenlægningen i adresse C002, det samme som 49154. Det kan meget let kaldes frem med et P, som lister programmet igen. Du vil se denne udskrift for linje C002:

C002	49154	0C	12
------	-------	----	----

Tidligere stod der

C002	49154	00	0
------	-------	----	---

Vi bad om at få lagt 7 og 5 sammen, og det er netop hvad der er sket. Og svaret, 12, *er* lagt ind i adresse C002.

Hvis du vil sikre dig at det ikke er noget tilfælde, kan du ændre indholdet i C000 og C001 og se om C002 indeholder den nye sum. Det kan for eksempel gøres ved direkte indskrivning af

POKE 49152,23 (el. lign.)

POKE 49153,11 (el. lign.)

```
SYS(49155)
```

```
PRINT PEEK(49154)
```

Så skulle du få resultatet 34, nemlig $23 + 11$. Prøv det flere gange med forskellige tal (lad os sige under 100) til erstatning for 23 og 11.

NB: VIGTIGT

Når du starter et maskinkodeprogram med SYS *skal* maskinkoden ende med kommandoen

```
RTS
```

(opkode 60 **Re**Turn from **S**ubroutine, 'retur fra underprogram'), som i dette tilfælde bringer dig tilbage til BASIC. Hvis du ikke gør det, vil computeren gladeligt kværne videre og opfatte det vās der ligger spredt i hukommelsen som om det var vaskeægte maskinkode – det stakkels kræ ved jo ikke bedre – og så *udføre det*. Resultatet bliver almindeligvis løjerligt, for at sige det mildt. Det er ikke usædvanligt at computeren æder sit eget program og på den måde begår en slags elektronisk harakiri.

Det er faktisk ikke nogen dårlig idé at ændre lidt ved LOADER, så programmet hæfter en kode 60 bag efter alle maskinkoder du skriver, bare for det tilfældes skyld at du har glemt det (et ekstra RTS kan ikke gøre skade). Erstat linje 360 med

```
360 IF H$="S" THEN POKE AD,96:GOTO 90
```

EDIT-FUNKTIONEN

For at gøre testning lettere – og for at gøre det muligt for dig at rette fejl – vil vi tilføje en EDIT-rutine til LOADER. Det er en yderst beskedne rutine: den giver dig bare mulighed for at ændre indholdet i en adresse, og at gøre det igen hvis du har lyst. Vil du have en mere raffineret editor, så kig i kapitel 21 (hvor det faktisk er 64'erens egen BASIC-editor vi bruger).

```
1000 PRINT:PRINT "EDIT":PRINT
1010 INPUT "ADRESSE I DECIMAL";AD
1020 PRINT "NYT INDHOLD I HEX"
1030 GOSUB 1300:H$=Q$:PRINT H$;
1040 GOSUB 1300:L$=Q$:PRINT L$
1050 GOSUB 600
1060 POKE AD,DK
1070 PRINT "MERE?"
1080 GOSUB 1300
1090 IF Q$="S" THEN 100
1100 GOTO 1010
```

Når du har skrevet det, så RUN, tryk E for EDIT og indtast

```
49152      17
```

Når du bliver spurgt om

```
MERE?
```

trykker du RETURN og indtaster

```
49153      0B
```

Igen får du spørgsmålet

```
MERE?
```

men denne gang stopper du ved at trykke

S

og kommer tilbage til menuen. Tryk R for RUN, så P for at udlæse resultatet. Kig på adresse C002. Den skulle gerne indeholde 22 (hex) lig med 34 (decimal). 17 hex er 23 decimal, og 0B hex er 11 decimal, og $23 + 11 = 34$. Så virkede det altså! Nu kan du med EDIT lægge andre tal ind, RUNne og PRINTE resultatet ud.

LAGRING AF MASKINKODE

Man kan ikke gemme maskinkode på tape eller disk lige så let som man kan gøre det med BASIC. Man bliver nødt til at skrive sine egne rutiner til det. En god udvej er at bruge *filer* – se mere i *Programmer's Reference Guide*. Her får du nogle rutiner som du kan tilføje til LOADER-programmet.

Først må du udvide valgmulighederne:

```
115 PRINT "F:FILE  I:INPUT"
172 IF Q$="F" THEN 1500
174 IF Q$="I" THEN 1700
```

Så skal du tilføje en rutine som kan lagre data i en fil:

```
1500 PRINT "HAR DU DET RIGTIGE BAAND PAA?"
1510 INPUT "FILENS NAVN";F$
1520 OPEN 1,1,1,F$
1530 INPUT "BEGYNDELSSESADRESSE";BA
1540 INPUT "KODENS LAENGDE";KL
1550 FOR T=BA TO BA+KL-1
1560 Y=PEEK(T)
1570 PRINT #1,Y
1580 NEXT
```

```
1590 CLOSE 1
1600 GOTO 100
```

Hvis du har diskettestation i stedet for kassetteoptager skal linje 1520 udskiftes med

```
1520 OPEN 1,8,2,F$+",SEQ,W"
```

Nu kommer input-rutinen:

```
1700 INPUT "NAVN PAA FIL DER SKAL FINDES";F$
1710 OPEN 1,1,0,F$
1720 INPUT "BEGYNDELSSESADRESSE";BA
1730 INPUT "KODENS LAENGDE";KL
1740 FOR T=BA TO BA+KL-1
1750 INPUT #1,X
1760 POKE T,X
1770 NEXT
1780 CLOSE 1
1790 GOTO 100
```

Hvis du har diskette skal du ændre:

```
1710 OPEN 1,8,2,F$+",SEQ,R"
```

Lad os sige du har skrevet 100 bytes kode ind fra adresse 49152. Du kan gemme koden på bånd ved at trykke F for FILE. Kassetteoptageren skal være tilsluttet, det er klart.

Først bliver du spurgt om hvilket navn du vil give filen. Skriv navnet, du vil måske kalde den

HANS

Båndet snurrer mens header-blokkken lægges ned på det. Det standser. Du vil så blive spurgt om begyndelsesadressen; skriv

49152

Du kan fikse op på programmet så det automatisk bruger samme begyndelsesadresse som i LOAD-rutinen, men for fleksibilitetens skyld er det rart at kunne ændre det. Derpå bliver du spurgt om kodens længde (også her kan du fikse programmet op), og du skriver

100

Nu ruller båndet igen; ved længere programmer vil du opdage at det standser og starter flere gange (det skyldes den måde kassettebufferen virker på).

Når du vil indlæse programmet fra bånd skal du vælge 'T' og gentage de samme trin.

FØR DU GÅR VIDERE

Gem den færdige version af LOADER på kassette med
SAVE "LOADER"

for fra nu af skal vi bruge den til indskrivning af alle
vore maskinkodeprogrammer

FLERE TEST

Du *behøver* ikke bruge LOADER for at udføre maskinkode. Det kan gøres med et ganske almindeligt SYS(49155). Du kan for eksempel teste det hele meget hurtigere med et BASIC-program:

```
2000 PRINT "SAMMENLÆGNING AF 8 BITS TAL -  
      TESTROUTINE"
```

```
2010 INPUT "INDHOLD TIL C000";M
```

```
2020 INPUT "INDHOLD TIL C001";N
```

```
2030 POKE 49152,M;POKE 49153,N
2040 SYS(49155)
2050 PRINT "C002 INDEHOLDER",PEEK(49154)
2060 GOTO 2010
```

Start det med GOTO 2000 og leg med det så længe hjertet begærer.

OVERLØB

Jeg bad dig bruge tal under 100. Man skal altid være mistænksom over for den slags halve forklaringer. Hvad sker der hvis vi beder LOADERS 8-bit-sammenlægger om at lægge 200 og 200 sammen? Hvilket svar regner du med at få?

Det svar du får er 144! Er maskinen da blevet vanvittig?

Ikke det fjerneste. Problemet er, som jeg har understreget det hele tiden, at vi har bygget en 8-bit-sammenlægger. Enhver mente der overføres til den niende bit (når vi har rundet de 256) er simpelt hen gået tabt. Du vil se at $144 + 256$ giver 400, som er det korrekte svar. Husker du kilometertælleren fra kapitel 3? Det er det samme der sker her.

Dette fænomen hedder *overløb* eller *overflow*. Det er noget som programmøren skal tage sig af, hvis det har nogen betydning. For når jeg tidligere sagde at menten er 'gået tabt', så var det ikke helt rigtigt. Der er et felt i processorstatusregistret som computeren kan bruge til at undersøge om der er indtruffet overløb. Det kan programmøren udnytte til at foretage de nødvendige skridt for at bringe udregningen på rette spor igen. Jeg nævner det her, bare for igen at indskærpe at maskinkode overlader det meste tænkearbejde til dig.

Når man skriver et maskinkodeprogram kan man gøre sig livet lettere ved at bruge noget lidt mere handy end tocifrede hexkoder!

8 · ASSEMBLERSPROG

Hvis du prøver at skrive et maskinkodeprogram har du rigeligt at tænke på uden også at skulle huske på alle opkoderne i hexadecimaltal. Det er eksempelvis meget lettere at tænke “Gem akkumulatorens indhold i lageret” end at huske opkoden 8D. Der findes et systematisk sæt *mnemonics*, huskekoder til hjælp for programmøren. Mnemonic i det nævnte eksempel hedder

STA

(‘**ST**ore **A**ccumulator’), og det er en hel del mere overskueligt.

Programmøren udarbejder derfor almindeligvis sit program med mnemonics, og først når han er tilfreds omsætter han det til hexkode (eller får et særligt program, kaldet en *assembler* til at gøre det). Man siger om programmer skrevet med mnemonics, at de er skrevet i *assemblersprog*.

Her kommer 8-bit-sammenlæggeren i assemblersprog. Først må vi fastlægge disse områder i hukommelsen:

C000 Data: første tal
C001 Data: andet tal
C002 Data: summen anbringes her
C003 Programstart

Og så programmet:

CLC	(Slet flaget for mente)
LDA C000	(Indlæs værdien fra C000 i akkumulatoren)
ADC C001	(Addér med værdien fra C001 (med mente))
STA C003	(Gem akkumulatorens værdi i C003)
RTS	(Retur fra maskinkode-underprogram)

(De tre nye mnemonics står som forkortelser for: '**CL**ear **Car**ry flag', '**LoaD** **A**ccumulator' og '**AD**d with **Car**ry'). Denne måde at skrive maskinkode på er meget nemmere at følge med i – især med lidt øvelse!

I dette kapitel vil jeg vise dig et antal enkle eksempler, programmer der udfører aritmetiske operationer som vi let kan efterkontrollere. Jeg skriver dem i mnemonics, forklarer hvad de gør og omsætter dem til hex. Din opgave er så at bruge **LOADER** til at få dem ind i hukommelsen, udføre dem og kontrollere at de har gjort det rigtigt (vær omhyggelig med databytes). En mere systematisk gennemgang af de eksisterende mnemonics og deres opkoder vil jeg gemme til senere kapitler.

SUBTRAKTION

Mnemonic for subtraktion er

SBC

som står for '**SuB**tract with **Car**ry', fratræk med mente. At trække fra med mente vil sige at 'låne'; hvis der er overført en mente fra en tidligere operation skal den regnes med i subtraktionen. Af samme grund hedder mnemonic for addition også **ADC**, ikke bare **ADD**; også her regnes der med

mente. For at undgå at vi får besvær med uvedkommende
menter, justerer vi dem før vi bruger ADC eller SBC. Den
eneste reelle faldgrube er, at mens man skal bruge CLC
(**CL**ear **Car**ry, 'slet mente') foran ADC, så er det rigtige for-
an SBC at bruge den nye kommando SEC (**SE**t **Car**ry, 'sæt
mente') med opkoden 38. Det skyldes at processorens carry-
flag fungerer lidt underligt (se kapitel 10).

Programmet vil virke på nøjagtig samme måde som før:
vi gemmer de to tal i C000 og C001 og differencen mellem
de to i C002. CPU'en skal

Sætte carry-flaget

Indlæse værdien fra C000 i akkumulatoren

Derfra trække værdien fra C001

Gemme resultatet i C002

Vende tilbage til BASIC

I assemblersprog har vi:

SEC

LDA C000

SBC C001

STA C002

RTS

Nu omregner vi til hex, idet vi tager tillæg 4 til hjælp:

Assemblersprog	Hex
SEC	38
LDA C000	AD 00 C0
SBC C001	ED 01 C0
STA C002	8D 02 C0
RTS	60

Det var min del af arbejdet. Så er det din tur: jeg vil bede dig bruge dette sammen med LOADER til at udregne $114 - 75$ (decimal). Se om du kan klare det selv *inden* du læser videre.

Her kommer løsningen på hvordan jeg ville have dig til at gøre det.

Først skal du ved hjælp af tillæg 1 finde ud af hvad 114 og 75 hedder i hex. De hedder 72 og 4B. Vælg så LOADERens mulighed L for at lægge data og program ind i hukommelsen, og brug standardbegyndelsesadressen (default-værdien) samt 3 databytes. Koden (data og program) skal lægges ind i denne rækkefølge

72 4B 00

data

38 AD 00 C0 ED 01 C0 8D 02 C0 60

program

Afslut med S for at komme tilbage til menuen. Tryk så P for at kontrollere at du har gjort det rigtigt og S for at gå ud af programmet igen; tryk endelig R for at udføre maskinkoden og P for at undersøge hvad der står i C002. Hvis alt er i skønneste orden skal der stå 39 (27 hex).

SAMMENTÆLLING AF FLERE TAL

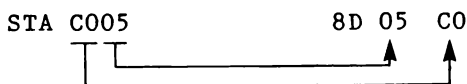
Lad os med det samme udvalg af kommandoer løse en lignende opgave: at addere en række på fem tal, som ligger i C000-C004, og lægge resultatet i C005. Selve programmet begynder altså i C006. Ingen grund til panik – her har du koden, dels i mnemonics, dels oversat til hex:

Assemblersprog	Hex
CLC	18
LDA C000	AD 00 C0
ADC C001	6D 01 C0
ADC C002	6D 02 C0
ADC C003	6D 03 C0
ADC C004	6D 04 C0
STA C005	8D 05 C0
RTS	60

Denne gang er der 6 databytes: 5 til tallene og 1 til resultatet. Brug LOADER til at skrive det hele ind med, og vælg selv tal der skal lægges sammen, men husk på at en total på over 255 vil savne en eller flere menter. Jeg ville foreslå at du holder alle tal under 50 decimal (32 hex) for ikke at løbe ind i vanskeligheder.

EN SÆRHED VED 16 BIT

Måske har du bemærket et system i hvordan adresserne noteres i opkoderne for LDA, ADC, SBC og STA. Da jeg for eksempel ville gemme akkumulatorens indhold i C005 brugte jeg opkoden:



Opkodens anden og tredje byte er de to adressebytes – men *i omvendt orden*.

Det er en uomgængelig regel for 6510-processoren. I opkoder der indeholder 2-bytes adresser står de to bytes altid i *modsat rækkefølge* af hvad de gør i adressen. Altså

Lille byte først, så store byte.

Det er ikke noget problem når man først er vant til det, men man skal passe på.

ADDITION MED MENTE

Vi skal nu se hvordan man behandler menter. Lad os skrive en 16-bit-sammenlægger. Dataområdet vil se sådan ud:

C000		Første tal, lille byte
C001		Første tal, store byte
C002		Andet tal, lille byte
C003		Andet tal, store byte
C004		Sum, lille byte
C005		Sum, store byte
C006		Her begynder programmet

De vigtigste skridt vil være:

Slet carry-flaget

Indlæs første tals lille byte i akkumulatoren

Læg dertil andet tals lille byte

Gem resultatet i summens lille byte

SLET IKKE CARRY-FLAGET DENNE GANG

Gentag processen med store bytes

Ved ikke at slette carry-flaget sikrer vi os at menten, hvis der er nogen efter den første addition, *bliver* regnet med i den anden.

Her er programmet i assemblersprog og hex:

CLC	18
LDA C000	AD 00 C0
ADC C002	6D 02 C0
STA C004	8D 04 C0
LDA C001	AD 01 C0
ADC C003	6D 03 C0
STA C005	8D 05 C0
RTS	60

LOAD programmet med 6 databytes og prøv det. Vil vi for eksempel lægge 30669 (decimal) til 17391 (decimal) omsætter vi dem til hex og får 77CD og 43EF. Så skal vi lægge disse bytes ind som data (og nuller i de to sidste datafelter) på denne måde:

Adresse	Indhold
C000	CD
C001	77
C002	EF
C003	43
C004	00
C005	00

Når programmet udføres får vi resultatet:

C004	BC
C005	BB

og BBBC (hex) er 48060 (decimal), hvilket er korrekt.

Prøv at sætte en ny CLC-kommando ind efter det første STA i programmet. Vi får så BABC som svar, hvilket er 47804. Det er 256 for lidt – og det er den manglende mente der er synderen!

Selv om vi har taget vare på *denne* mente er der alligevel

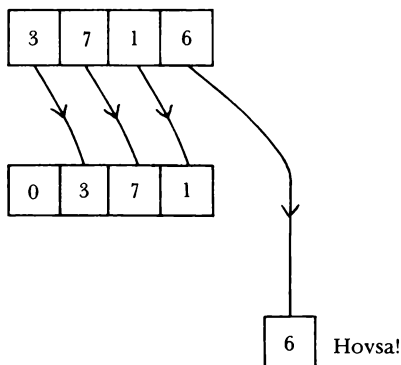
en anden mente, som vil optræde hvis totalen runder 65535 (FFFF hex), og den går tabt i det eksisterende program (man kunne overveje at udvide dataområdet med en byte på C006 som kan opbevare den sidste mente – hvis der er nogen). Vi får at se nøjagtig hvordan menten virker når vi kommer ind på *flag* i kapitel 10.

HALVERING

Regning med tiere er som regel nem at foretage i decimal-systemet; regning med toere er tilsvarende nemt i binær- eller hexsystemet.

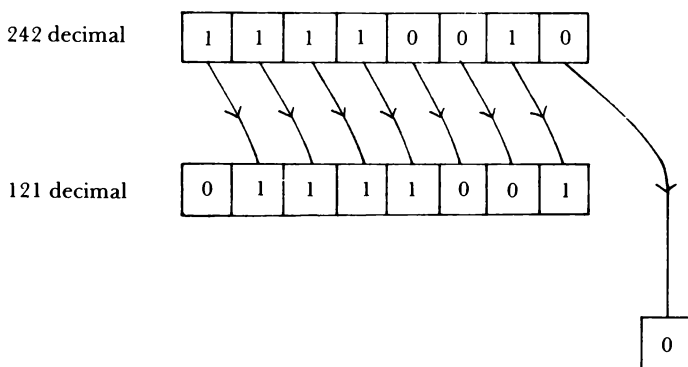
Tænk decimalt et øjeblik – hvis du kan huske hvordan man gør! Hvordan dividerer man fx 3710 med 10? Man smider selvfølgelig det sidste ciffer væk og får 371. Det går også glimrende med et tal som 3716; divisionen giver nøjagtig 371,6, og hvis du er indstillet på at undvære decimalbrøken (at runde *ned*) får du 371, som igen er tallet minus sidste ciffer.

Med andre ord er tallet *skubbet* en plads til højre, hvorved cifret længst til højre falder af, sådan:



Det lille 0 jeg har puttet foran gør ingen skade – det sørger bare for at rubrikkerne er fyldt ud.

I binærsystemet sker der det samme med toere som der i decimalsystemet sker med tiere. I binær kan vi dividere med to – altså *halvere* et tal – ved at skubbe dets cifre en plads til højre (hvis det oprindelige tal er ulige, vil brøken $1/2$ (som i binær hedder 0,1) gå i vasken). Lad os lige afprøve det på tallet 242 (decimal), som hedder 11110010 i binær. Det gør vi her:

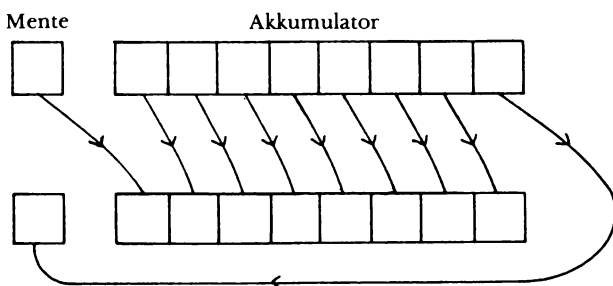


Resultatet er 01111001 eller 121 decimal. Sådan!

Der findes en 6510-kommando 'forskyd akkumulatorindholdet til højre'; dens mnemonic er

ROR (**RO**tate **R**ight)

med opkode 6A. Hvis der er blevet en mente tilovers fra en tidligere operation flyttes denne til bittens længst til venstre (og den bit som jeg sagde gik tabt, ender altså faktisk i carry-feltet):



Det er somme tider hvad man har brug for; ellers kan et kvikt lille CLC hurtigt fjerne cifret.

Lad os til eksempel lægge et tal ind i C000 og halvdelen af det ind i C001 (men udelade den overskydende 1/2 hvis tallet er ulige):

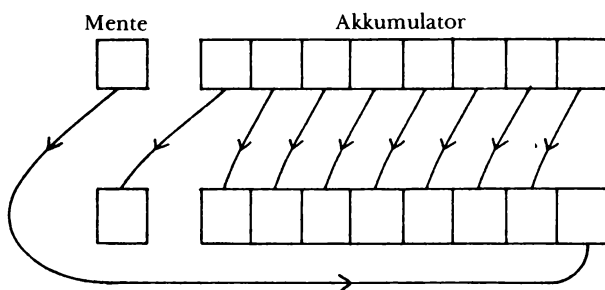
CLC	18
LDA C000	AD 00 C0
ROR	6A
STA C001	8D 01 C0
RTS	60

Skriv dette ind sammen med to databytes og prøv det.

FORDOBLING

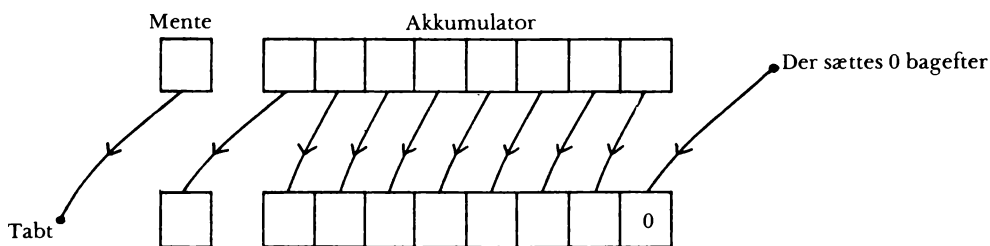
For at fordoble et tal skal vi rykke det til *venstre*. I et anfald af gavmildhed har konstruktørerne af 6510 forsynet os med *to* forskellige måder at gøre det på. Kun hvad menten angår virker de forskelligt. Den ene hedder 'rotér til venstre':

ROL (**R**Otate **L**eft, opkode 2A)



Den anden hedder 'aritmetisk venstreforskydning':

ASL (**A**rithmetic **S**hift **L**eft, opkode 0A)

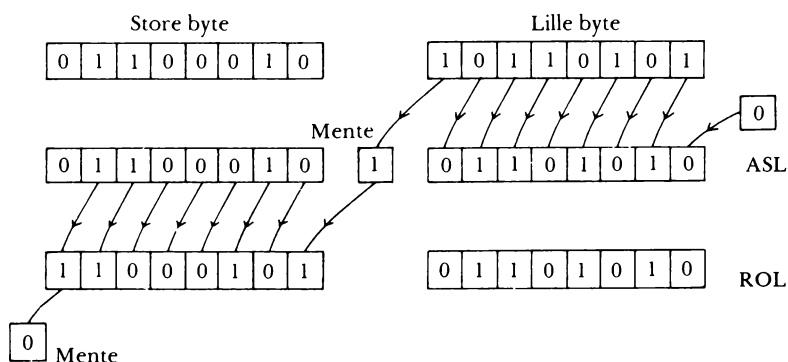


Vi kan fordoble et 1-bytes tal (under 128 for at undgå men-
teproblemer) beliggende i C000 og lægge resultatet i C001
på denne måde:

```
LDA C000    AD 00 C0
ASL         0A
STA C001    8D 01 C0
RTS        60
```

(Der er ingen grund til at bruge CLC denne gang – hvor-
for?). Skriv dette ind sammen med to databytes og kontrol-
lér at jobbet bliver udført.

Men for at fordoble et 16-bits (2-bytes) tal skal vi bruge ASL på lille byte og ROL på store, fordi vi *gerne* vil have den første mente til at rykke op:



Som sædvanlig vil jeg lægge tallene i C000-C001 og gemme det fordoblede tal i C002-C003 (først lille, så store byte). Koden er

```
LDA C000      AD 00 C0
ASL           0A
STA C002      8D 02 C0
LDA C001      AD 01 C0
ROL           2A
STA C003      8D 03 C0
RTS           60
```

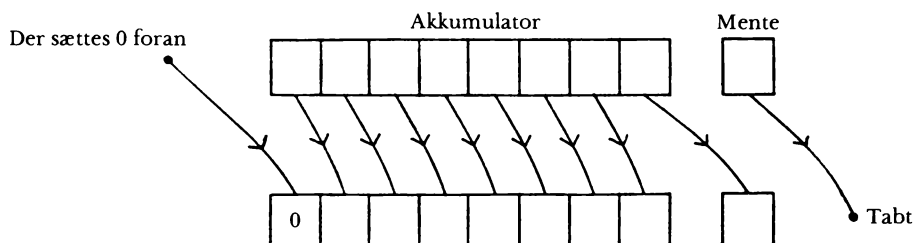
Indskriv det med 4 databytes og test det som sædvanlig.

EN ANDEN SHIFT-KOMMANDO

Der er endnu en kommando i dette tankeunivers, som arbejder sammen med ROR på samme måde som ASL gør med ROL. Den hedder 'logisk højreforskydning':

LSR (Logic Shift Right)

Lige som ROR foretager den en forskydning til højre, men den lægger et 0 i bit 7. Den halverer altså en enkelt byte, uden at det er nødvendigt at slette menten først.



Se, jeg sagde jo jeg ikke altid ville fortælle hele sandheden . . .

9 · ADRESSERINGSMÅDER

6510 er et mere alsidigt væsen end jeg hidtil har bildt dig ind. Mange af dens instruktioner kan benyttes på *mange forskellige måder*, som kaldes adresseringsmåder og har hver deres opkode. Hvordan man tæller dem afhænger af hvor man vælger at sætte grænsen; jeg får det til 12 forskellige adresseringsmåder alt i alt, men andre mennesker, som er mere nøjeregnende, kommer op på 13.

Eksempler viser bedst hvad der sker. Lad mig først tage vore gamle venner LDA og STA til hjælp.

UMIDDELBAR ADRESSERING

bruger man til at hente et bestemt *tal* ind i akkumulatoren (eller at foretage en operation med et tal). For at læse værdien 7D (hex) ind i akkumulatoren skal man bruge

LDA #7D A9 7D

Det er en 2-bytes opkode. Den første byte, A9, siger til computeren: 'Indlæs umiddelbart i akkumulatoren'. Nu *ved* den at den næste byte, 7D, er *det tal der skal læses ind*.

Tegnet # i mnemonic minder *programmøren* om at det er tallet 7D sagen drejer sig om, ikke en adresse der hedder 7D. Tegnet kaldes 'nummertegn', fordi det i USA bruges med samme betydning som vi bruger 'nr.'.

STA kan ikke bruges med umiddelbar adressering; og det

er i grunden ret indlysende når man tænker over det. Det eneste sted man kan lagre noget er jo i en adresse.

ABSOLUT ADRESSERING (= IKKE-ZEROPAGE)

Denne adresseringsmåde er den som vi hele tiden ubekymret har brugt ved LDA. Den lader akkumulatoren med indholdet af den adresse som er specificeret i kodens to næste bytes (i rækkefølgen lille byte/store byte). Skal vi lade akkumulatoren *fra*, det vil sige *med indholdet af* adresse C051, skal vi bruge

```
LDA C051      AD 51 C0
```

På tilsvarende vis skal vi, for at lagre akkumulatorindholdet i adresse C051, bruge

```
STA C051      8D 51 C0
```

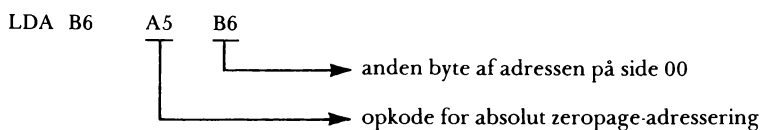
AD i det første eksempel betyder altså 'lad akkumulator absolut', og 8D i det andet eksempel betyder 'lagr akkumulator absolut'.

Opkodens tredje byte er store byte i adressen, og som du husker fra kapitel 4 er det det samme som *sidenummeret*. Med denne adresseringsmåde skal sidenummeret være forskelligt fra nul, for der findes nemlig en særlig måde at adressere zeropage (side nul) på, som – sjovt nok – går under navnet . . .

ZEROPAGE-ADRESSERING

Hvis du vil anvende absolut adressering på side 00 (adresserne 0000–00FF hex, 0–255 decimal) kan du *udelade* store byte, altså 00. Men så bliver opkoden en anden. Hvis du for

eksempel skal lade akkumulatoren fra adresse 00B6 på side nul, skriver du



Og for at lagre akkumulatorindholdet i adresse 00B6 skal du bruge

STA B6 85 B6

Side nul er i særdeleshed nyttig hvis man starter med en 'uberørt' 6510 (hvilket *ikke* er tilfældet!), fordi det sparer RAM-plads at udelade den overflødige byte 00. Det ved de folk der har forfattet Commodore 64's operativsystem, og de snøbler har hugget næsten hele side nul! Men de *har* dog efterladt os almindelige dødelige sølle *fire* symbolske bytes på side nul, nemlig på adresserne

00FB
00FC
00FD
00FE

Hvis du vil gøre brug af side 0 samtidig med at du bruger BASIC må du stuve alting ind der.

IMPLICIT ADRESSERING OG AKKUMULATOR- OPERATIONER

Nogle operationer berører udelukkende akkumulatoren – og der er nogle der overhovedet ikke berører den! Eksemp-

ler på det første er (den ene af opkoderne for) ROR og ROL. Vil man forskyde akkumulatorindholdet til højre er det nok at skrive

ROR

6A

uden *nogen* tilføjelser af adresser eller tal til opkoden.

Et eksempel på det andet er

RTS

60

som vi har brugt til at bringe os tilbage til BASIC (men mere generelt udtrykt bringer det os tilbage fra en underrutine til hovedprogrammet; se kapitel 15).

Her i denne bog vil begge disse adresseringsmåder som kun består af én byte blive betragtet som 'implicit adressering'; det vil sige som 'adresseringsmåden' der ingen adresse har!

ANDRE ADRESSERINGSMÅDER

De resterende otte måder er noget mere komplicerede. Det er *indirekte* og *relativ* adressering (som bliver beskrevet i kapitel 11 under forgrening og hop) samt seks *indekserede* adresseringer (se kapitel 13 og 14 om indekseret og indirekte adressering). I tillæg 4 ses alle de mulige adresseringer for hver enkelt kommando sammen med den tilsvarende opkode. Bemærk at mange kommandoer kun bruger en eller to adresseringsmåder, og der er ingen kommando der bruger dem alle.

ET EKSEMPEL

Her er der et lille eksempel som benytter alle de fire hidtil

forklarede adresseringer. Det er lidt konstrueret, men det kan kaste lys over de resterende problemer.

1. Tænk på et tal, fx 43 decimal, 2B hex; gem det i 00FB på side 0.
2. Gang det med to (bemærk at tallet bliver liggende i akkumulatoren; STA lægger en *kopi* på det ønskede sted, mens originalen forbliver intakt).
3. Læg dertil 17 decimal, 11 hex.
4. Gem resultatet i C000, *ikke* på side nul.

Adresseringsmåde	Mnemonic	Opkode	Antal bytes
Implicit	CLC	1 8	1
Umiddelbar	LDA #2B	A 9 2 B	2
Zeropage	STA FB	8 5 FB	2
Implicit	ROL	2 A	1
Umiddelbar	ADC #11	6 9 1 1	2
Absolut (ikke zeropage)	STA C000	8 D 0 0 C 0	3
Implicit	RTS	6 0	1

(Hvis du godt vil afprøve det, så husk at reservere en data-byte C000 foran programområdet).

Læg mærke til at man kan se på mnemonicinstruktionens opbygning hvilken adresseringsmåde der er tale om:

Implicit:	CLC	(uden tilføjelser)
Zeropage:	LDA FB	(én byte tilføjet)
Umiddelbar:	LDA # 2B	(# plus én byte tilføjet)
Absolut:	STA C000	(to bytes tilføjet: C0 og 00)

De otte adresseringsmåder som bliver forklaret senere har også hver deres opbygning. Bemærk at det kun er program-

møren der har fornøjelse af at mnemonics er forskelligt opbygget; computeren interesserer sig kun for opkoden. Du kan opfinde dit eget sæt mnemonics om du har lyst. Men de mnemonics der er foreskrevet af fabrikanten er industristandard, så (a) det vil være nemmere for dig at læse andre menneskers kode hvis du holder dig til standard, (b) det vil være nemmere for andre mennesker at læse din, og (c) hvis du køber et assemblerprogram vil det næsten med garanti bruge standarden.

Mnemonics i denne bog afviger på ét punkt fra standard; det er almindeligt at sætte symbolet \$ foran et hexstal, altså at skrive

\$F7

i stedet for bare

F7

Jeg er af den mening at det på dette sted vil være lettere at bruge hex som standard hele vejen igennem (og således undgå eventuelle uheldige forvekslinger af hex med decimal). Jeg har vrøvl nok med mine pengesager til daglig og vil gerne være fri for at strø dollartegn ud over mine programmer! Men du skal være klar over at dollartegnene kan være obligatoriske andre steder (i de fleste assemblerprogrammer i handelen, for eksempel). *Nu er du advaret!*

Et fleksibelt program må være i stand til at fungere forskelligt under forskellige forhold. 6510 registrerer konstant vigtige processortilstande og hvordan de påvirkes af den seneste operation, idet den løbende justerer processorstatusregistrets cifre.

10 · FLAG

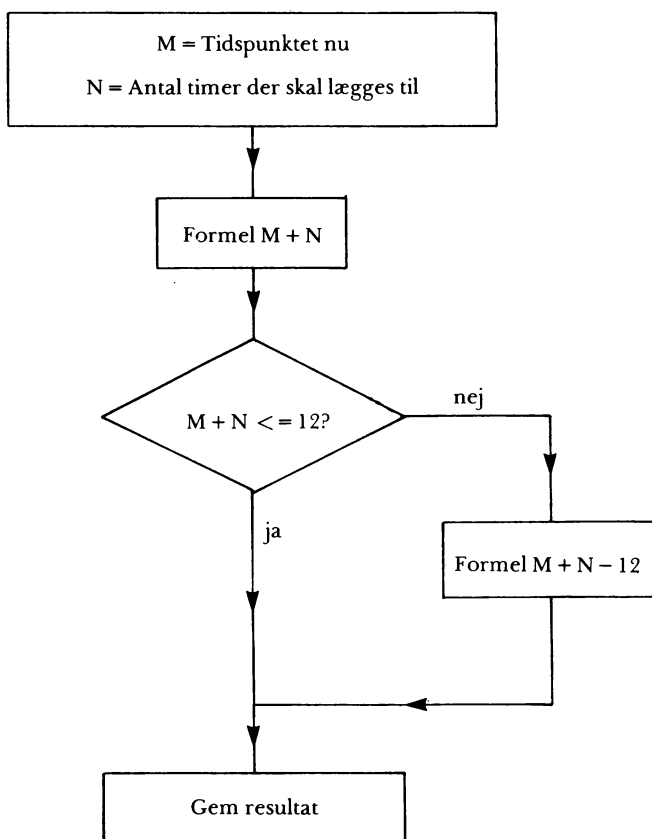
En helt grundlæggende computerteknik, som i hvert fald går tilbage til Charles Babbages ideer omkring 1830, er at få processtrømmen til at *forgrene* sig under bestemte forudsætninger. IF . . . THEN-kommandoen i BASIC udfører denne funktion.

Tænk for eksempel på når man skal lægge klokkeslæt sammen. Så er klokken 1 plus 7 timer lig med kl. 8, ligesom ellers. Men kl. 9 + 7 timer = kl. 4, ikke kl. 16 på et almindeligt 12-timers ur! Man kan ikke bare lægge tallene sammen. Processtrømmen er vist i blokdiagrammet på næste side.

Læg mærke til at vi bruger kl. 12, ikke kl. 0! I BASIC kan det gøres på denne måde:

```
10 LET M= et eller andet
20 LET N= et eller andet andet
30 S=M+N
40 IF S<=12 THEN 60
50 S=S-12
60 PRINT S
```

Jeg har med vilje brugt GOTO-muligheden her – ihvorvel ‘GOTO’ er underforstået – i stedet for en mere ‘struktureret’ løsning, fordi det giver holdepunkter for maskinkode, som langt fra er ‘struktureret’!



PROCESSORSTATUSREGISTRET

Det fundamentale spørgsmål ved forgrening er at afgøre om et bestemt tal (her $M + N - 12$) er positivt, nul eller negativt. Det er her P-registret, hvis imponerende navn pryder dette afsnit, kommer ind i billedet. Så nu kan jeg ikke længere knibe uden om, men må fortælle hvad det går ud på.

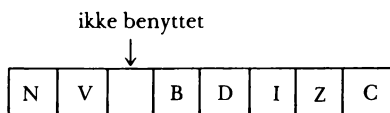
Hver enkel af P-registrets otte bit bruges som et *flag*. Det vil sige at det pågældende ciffer enten er

sat (= 1)
 eller *slettet* (= 0)

afhængigt af om en bestemt tilstand er indtruffet eller ikke indtruffet på det tidspunkt. De fleste operationer ændrer på flagenes mønster; en oversigt findes i tillæg 5.

I virkeligheden er der kun syv flag i P-registret, for én bit er 'reserveret til senere brug', hvilket er en pæn måde at sige på: 'vi kunne ikke blive enige om hvad vi skulle gøre med den'. Og nogle af disse syv har ikke meget interesse undtagen for hardware-entusiaster. Så det er ikke så vanskeligt.

Flagene er arrangeret på denne måde:



Jeg vil fortælle om deres betydning i en passende (rodet) rækkefølge.

Z: Zeroflag (nulflag)

Nulflaget sættes (= 1) hvis resultatet af en aritmetisk eller logisk operation er lig nul og slettes (= 0) hvis resultat er forskelligt fra nul. (En lille pudsighed: hvis resultatet er nul er flaget *ikke* nul, og omvendt. Programmering kan gøre én vanvittig. Vanvittig siger jeg dig! Meningen er den at cifret 1 i et flag betyder 'vift med flaget', 'den pågældende tilstand er indtruffet'. Og 'den pågældende tilstand' er i dette tilfælde nul.)

N: Negativflag (fortegnsflag)

Hvis resultatet af en aritmetisk operation bliver negativt sættes negativflaget (1); hvis det bliver positivt eller nul slettes det (0).

Eller det er hvad der *burde* ske. Desværre er det ikke helt sandt!

Det *er* sandt hvis du tænker på et 8-bits tal som et 7-bits tal med en fortegnsbite, som det blev omtalt i kapitel 3. Det vil altså sige: hvis din talrække tæller

0, 1, 2, . . . , 126, 127

men så skifter over til

-128, -127, -126, . . . , -3, -2, -1

i stedet for at fortsætte fra 128 til 255.

Så det der *faktisk* sker er at N-flaget meddeler dig indholdet af den bit som står *længst til venstre* i resultatet (bytes fra 0 til 127 begynder med 0-et-eller-andet, resten begynder med 1-et-eller-andet). Jeg vil udsætte yderligere kommentarer til vi får brug for dem.

C: Carry-flag (menteflag)

Hvis en addition resulterer i en mente, eller en subtraktion resulterer i at man må låne, signalerer carry-flaget denne begivenhed på sin egen besynderlige facon:

1. Hvis resultatet ved en ADC-instruktion overskrider 255, så er der en mente, og carry-flaget sættes (1). Ellers slettes det (0).
2. Hvis resultatet ved en SBC-instruktion underskrider 0, så må man låne, og carry-flaget *slettes* (0). Hvis der ikke skal lånes er carry-flaget *sat* (1).

Nu forstår du måske lidt af hvorfor jeg ikke havde lyst til at fortælle særligt meget om flag, ikke?

Det vigtigste er at holde hovedet koldt. Carry-flaget fortæller hvad der er sket, begge veje, men man skal huske

hvordan man skal forstå det den fortæller, afhængigt af om man har lagt til eller trukket fra. Her har du en lille tabel til hjælp (den forudsætter at du har M i akkumulatoren og til lægger eller fratrækker N):

Operation	Resultat i akkumulatoren	Carry-flagets status
ADC	$M + N$ (bortset fra mente)	0 hvis $M + N \leq 255$ 1 hvis $M + N \geq 256$
SBC	$M - N$ (bortset fra lån)	1 hvis $M - N \geq 0$ ($N \leq M$) 0 hvis $M - N < 0$ ($N > M$)

Det er underforstået at M og N som sædvanlig er 8-bits fortegnsløse tal mellem 0 og 255.

C- og Z-flagene vil meddele dig om M og N er lige store, om $M < N$ eller om $M > N$, hvis M og N er mellem 0 og 255.

Faktisk er C- og Z-flagene (lejlighedsvis sammen med N-flaget) de sandsynligvis eneste du vil bruge, medmindre du bliver virkelig seriøs. Men for fuldstændighedens skyld er der her en hurtig oversigt over de fire andre.

V: Overløbsflag (overflow)

Hvis du laver aritmetiske operationer med 8-bits tal forstået som 7-bits tal med fortegn (-128 til 127 igen, er du med?) virker dette flag nogenlunde som carry-flaget gør det ved almindelig 8-bit-aritmetik. Hvis resultatet går uden for området -128 til 127 , så sættes V-flaget (1).

Det kan også sættes udefra af et dertil egnet kredsløb, og det bruges til højst forskellige formål.

D: Decimalflag (decimalindstilling)

Der findes en anden type aritmetik ved navn *binærkodet decimal* (BCD), som repræsenterer et decimaltal *ciffer for ciffer* i hex. For eksempel repræsenteres

34125476

af disse bytes:

34 12 54 76

Men hvis du ville bruge dette tal som et gyldigt hexstal i en aritmetisk operation ville du komme ud i noget frygteligt rod; for eksempel virker carry-reglerne helt anderledes i BCD.

Ikke desto mindre bruges BCD somme tider på grund af dens slægtskab med decimal. Der er for eksempel mange lommeregnerne der anvender princippet. Hvis D-flaget er sat (1) vil 6510 udføre alle aritmetiske operationer som BCD-aritmetik.

Når du tænder for computeren bliver D-flaget automatisk slettet. Jeg vil foreslå at du lader det blive ved det!

I: Interruptflag (maskeret interrupt)

Interrupt er den måde eksterne enheder kommunikerer med CPU'en på. Hvis I-flaget er sat (1) er interrupt ude af drift (ikke mulig); hvis det er slettet (0) er interrupt i kraft (mulig).

B: Break-statusflag

Dette flag sættes (1) af BRK-kommandoen ('software break'). Så holder CPU'en op med at arbejde og venter på hjælp udefra. Det er et meget nyttigt flag for hele opbygningen af computeren, men for os har det meget ringe interesse.

KOMMANDOER SOM PÅVIRKER FLAGENE DIREKTE

Der er nogle kommandoer hvormed du kan slette flag (nulstille dem) eller sætte dem (på 1), uden at foretage andre ting. Det er

CLC	18	CL ear C -flag	SEC	38	SE t C -flag
CLD	D8	CL ear D -flag	SED	F8	SE t D -flag
CLI	58	CL ear I -flag	SEI	78	SE t I -flag
CLV	B8	CL ear V -flag			

Alle er 1-bytes opkoder med implicit adressering.

Nu hvor vi kender flagenes funktioner kan vi kaste et blik på hvordan de bruges til at kontrollere forgreninger i et program. Hermed introduceres en ny adresseringsmåde kaldet relativ adressering.

11 · FORGRENING OG HOP

Jeg omtalte tidligere at der findes en herligt enkel måde at få programmet til at hoppe fra én instruktion til en anden på. PC-registret (programtælleren – ikke at forveksle med P-registret, som indeholder flagene) indeholder adressen på den næste instruktion som skal udføres. Ved at ændre den adresse kan man narre PC til at omdirigere procesforløbet til en helt anden kommando.

Man kan ikke ændre PC-registret direkte, men der findes en hel stribe forgreningskommandoer hvormed man kan fremkalde de ønskede ændringer: BCC, BCS, BEQ, BMI, BNE, BPL, BVC, BVS. Hver af dem instruerer computeren om at kigge på et af flagene, se hvad det viser og, afhængig heraf, skubbe PC et passende stykke op eller ned – hvorved kontrollen overføres til den nye instruktion.

BEQ

De virker alle på samme måde, så når man først har forstået én er der ingen ben i de andre – bortset fra den lille detalje der hedder håndtering af flagene. Jeg vil begynde med BEQ, fordi det er lige ud ad landevejen, BEQ (**B**ran**ch** if **E**Qual) står for 'forgrening hvis lig med', men helt præcis

betyder det at der skal ske en forgrening hvis zero-flaget er sat.

Opkoden er på 2 bytes. Den første byte er F0. Den anden byte er et *offset*, en afstand mellem to adresser. Det behandles som et relativt tal (dvs. som et tal med fortegn: syv bit plus fortegn) mellem -128 og 127. Vi er stødt på det princip tidligere, men nu står vi ansigt til ansigt med det. Et positivt offset er en besked til PC-registret om at 'rykke så og så mange pladser fremad', og et negativt betyder 'ryk så og så mange pladser baglæns'. Jeg skal forklare det lidt mere omhyggeligt når vi har set på et eksempel.

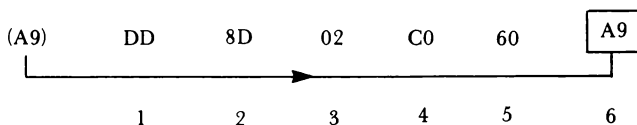
Lad os antage vi har to tal lagret i C000 og C001, og vi vil gerne se om de er lige store. Hvis de er det, vil vi lagre tallet EE i adresse C002. Hvis de ikke er lige store vil vi lagre DD der i stedet for. Sådan vil vi gøre det:

SEC	38	
LDA C000	AD 00 C0	Hent første tal M
SBC C001	ED 01 C0	Fratræk andet tal N
BEQ <i>spring</i>	F0 06	Forgrening til <i>spring</i> hvis M - N lig 0
LDA #DD	A9 DD	} Ellers lagres DD
STA C002	8D 02 C0	
RTS	60	Retur til BASIC
<i>spring</i> : LDA #EE	A9 EE	Hvis vi er havnet her er M - N lig med 0
STA C002	8D 02 C0	EE lagres i stedet for DD
RTS	60	Retur til BASIC

Lad os først se på hvordan det virker. Det første par instruktioner har vi set før. Når vi kommer til BEQ-instruktionen undersøger computeren zeroflaget. Hvis det er *sat* (hvilket betyder at den seneste operation der påvirkede flaget gav 0 som resultat - i dette tilfælde SBC-operationen) så skal programtælleren flyttes 6 frem (offset er 06, den anden byte i instruktionen).

Hvis vi begyndte med lige store værdier af M og N (i C000 og C001), hvis de fx begge to var 7B (hex), så ville resultatet af subtraktionen være nul og forgreningen ville ske (under forudsætning af at carryflaget først er blevet sat – og det er det).

På det tidspunkt hvor BEQ-instruktionen bliver aktuel står PC-registret allerede og kigger på den næste instruktion i rækken: LDA #DD med A9 som første byte i opkoden. Nu får det meddelelsen 'ryk 6 bytes længere frem'. Så det tæller videre i programmet fra A9:



som er begyndelsen på LDA #EE-instruktionen (på den linje som er mærket *spring*). Det er altså *denne* instruktion som computeren vil udføre nu. Den lagrer EE i adresse C002 og vender så tilbage til BASIC.

Hvis M og N derimod ikke var lige store, lad os sige M = 7B og N = C3, så ville zeroflaget ikke blive sat af SBC-kommandoen, og BEQ ville give besked til computeren om *ikke* at forgrene. Derved ville PC pege på den næste kommando på listen: LDA #DD. Når computeren fortsatte herfra ville den lagre DD i C002 og så være vendt tilbage til BASIC. (Bemærk at hver gren skal have sit eget RTS. Det der tæller er hvad computeren faktisk *udfører*, ikke hvad der ellers ligger og flyder i programmet. Hvis du udelader RTS i 'ikke lig med nul'-grenen ville programmet bare fortsætte til *skip* og videre derfra – og du ville ende med at have EE i C002 hvad enten du piber eller synger.)

RELATIV FORGRENING

Lad mig nu fortælle lidt mere om hvordan den korrekte offset-værdi beregnes i en forgreningskommando. I eksemplet ovenfor har jeg streget den under; den var 06. Hvorfor var den det?

Tænk på hvordan programmets bytes ligger i hukommelsen (se figur 1).

Sådan virker det med en *positiv* offsetværdi. Ved en *negativ* går man frem på samme måde, idet man stadigvæk går ud fra den position som PC ville have fortsat fra (kommandoen umiddelbart efter BEQ og offsetværdi), men nu tæller man baglæns:

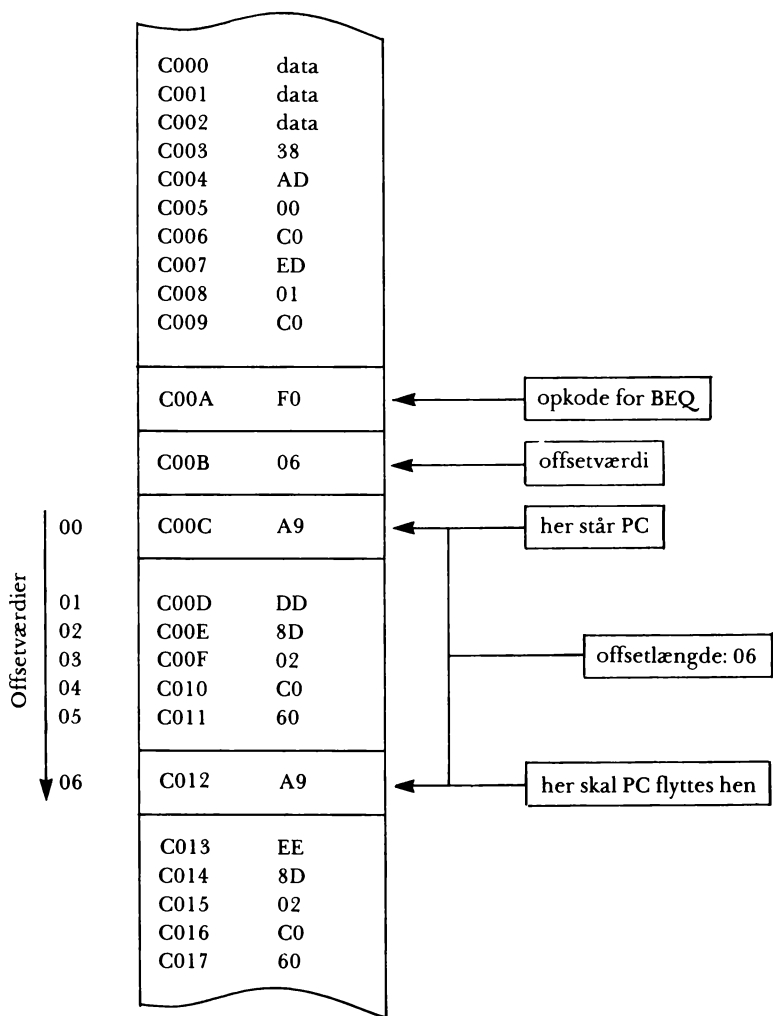
$-1, -2, -3, \dots$

indtil man kommer til den byte som indeholder *begyndelsen* på den ønskede opkode. Det er dog stadig nødvendigt at tage den fremkomne negativværdi og omsætte den til det binære relative tal tokomplementet, og derfra til hex. Heldigvis gør tillæg 1 det for dig.

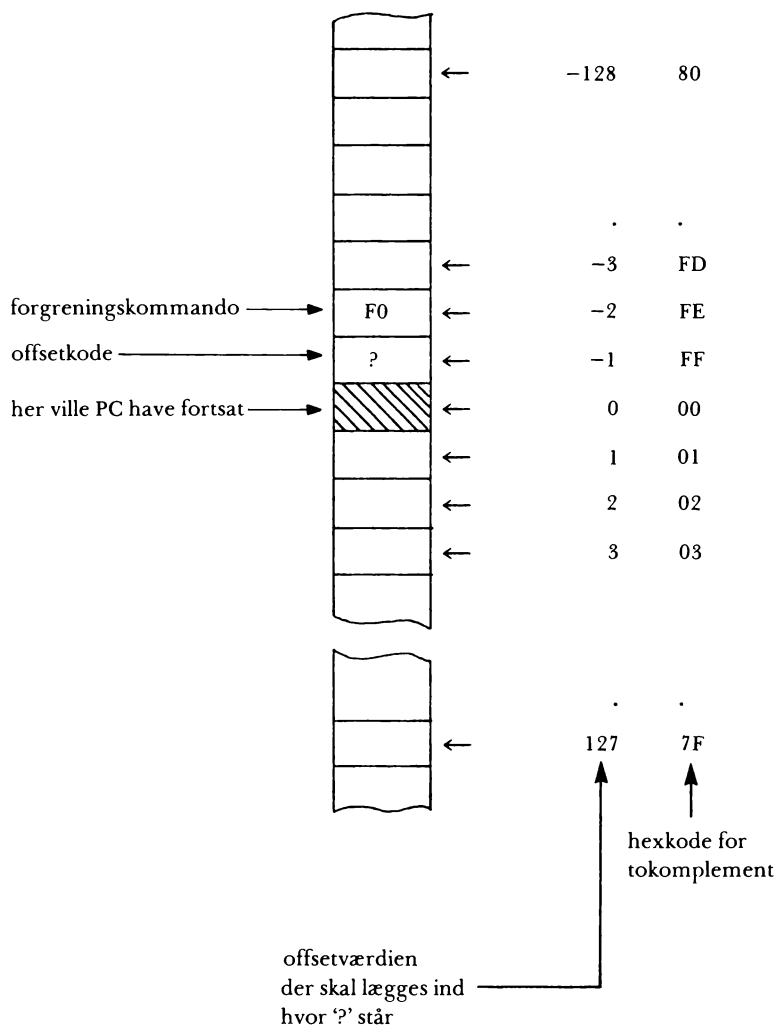
Hvis du fx vil foretage en forgrening på 37 bytes bagud, slår du -37 op i tillægget og finder DB. Det er offsetværdien. Figur 2 giver en samlet oversigt.

Jeg har gjort mig det til en regel at understrege alle relative hop-værdier i maskinkodelistninger.

Nu sidder I allesammen der og tænker: "Men hvad skal jeg gøre hvis jeg skal bruge en forgrening på mere end 127 bytes?" Du bliver faktisk nødt til at gøre det med flere små hop, eller også kan du bruge JMP-kommandoen, som beskrives senere. I praksis er svaret, at når du først skriver programmer hvor så store hop er *nødvendige*, så har du alligevel ikke brug for min hjælp til at finde ud af det.



Figur 1



Figur 2

LABELS

Når du skriver et program gider du ikke beskæftige dig med alt dette abracadabra. I stedet for lader du et felt stå åbent hvor offsetværdien skal stå (jeg foretrækker at lave en understregning for at minde mig selv om at der mangler en byte) indtil du har skrevet alle dele af programmet færdig. Gå så tilbage til det blanke felt, tæl antallet af bytes og udfyld feltet. (Bemærk at det er antallet af bytes, *ikke* antallet af instruktioner der udgør offsetværdien. Opkoderne har forskellig længde, og derfor er den enkleste løsning at tælle de eksisterende bytes.)

Til at identificere de instruktioner som du vil lave forgrening til bruges labels: små kvikke navne som *spring*, *loop5* o.l. Sæt labelnavnet foran instruktionen som vist, og henvis til det i assemblerkoden på det sted hvor offsetværdien skulle stå. For eksempel

```
    BEQ spring
    .
    .
    .
spring: LDA #EE
```

Med en god assembler kan du bruge labels; den vil selv udregne de relative offsetværdier.

ANDRE FORMER FOR FORGRENING

De andre forgreningskommandoer arbejder på nøjagtig samme måde med relativ adressering (offsetværdier). Den eneste forskel er hvilke flag de undersøger og hvordan de reagerer. Her er den komplette liste med opkoderne i parentes:

BCC (90)	B ranch if C arry C lear	hvis C-flaget er 0
BCS (B0)	B ranch if C arry S et	hvis C-flaget er 1
BEQ (F0)	B ranch if E Qual	hvis Z-flaget er 1
BNE (D0)	B ranch if N ot E qual	hvis Z-flaget er 0
BMI (30)	B ranch if M inus	hvis N-flaget er 1
BPL (10)	B ranch if P lus	hvis N-flaget er 0
BVC (50)	B ranch if o V erflow C lear	hvis V-flaget er 0
BVS (70)	B ranch if o V erflow S et	hvis V-flaget er 1

Alle har 2-bytes opkoder: den viste byte plus offsetværdien. Da de *kun* kan bruges med én adresseringsmåde har mnemonics altid den samme opbygning.

FORTEGNSTEST

Som eksempel på brugen af forgrening vil jeg skrive et program, hvormed du kan teste hvad jeg sagde tidligere om N- og C-flag. Grundideen er et problem som man tit møder i et program: der er givet to absolutte 1-bytes tal (dvs. tal uden fortegn mellem 0 og 255), fx M og N; afgør om $M > N$ eller ej.

Jeg vil som sædvanlig lægge de to tal M og N i C000 og C001. I C002 vil jeg sætte mit eget lille flag: 0 hvis $M - N$ er negativt, 1 hvis $M - N$ er positivt. Du vil straks få at se hvorfor.

Her er koden:

```

SEC                38
LDA C000           AD 00 C0
SBC C001           ED 01 C0
BCC neg           90 06      Offset for relativt hop
LDA #1            A9 01
STA C002          8D 02 C0
RTS               60
neg: LDA #0       A9 00

```

STA C002	8D 02 C0
RTS	60

Skriv det ind med 3 databytes; men *lad være* med at starte det denne gang. Afbryd LOADER-programmet ved at skrive S, og skriv så denne BASIC-rutine, som vil gøre det muligt at teste en masse muligheder:

```
5000 INPUT "M,N";M,N
5010 PRINT M;"-";N;"ER ";
5020 POKE 49152,M
5030 POKE 49153,N
5040 SYS(49155)
5050 X=PEEK(49154)
5060 IF X=1 THEN PRINT "POSITIVT"
5070 IF X=0 THEN PRINT "NEGATIVT"
5080 GOTO 5000
```

Det er en endeløs løkke, hvormed du kan teste alle værdier af M og N. Dog skal de være mellem 0 og 255.

Start med GOTO 5000 og se hvad der sker. Giver det nogen mening? Det skulle det gerne.

Lad os nu se på hvad der ville ske hvis du gjorde det mest 'indlysende' og brugte BMI (**B**ran**ch** if **M**inus) i stedet for BCC. Erstat BCC-linjen med

BMI neg	30 06
---------	-------

og brug LOADERS E-funktion til at gøre det med (en god skik!). Hvis du nu bruger BASIC-rutinen fra 5000 synes alt vel så længe du bruger tal som 100 – 68, der betragtes som et positivt resultat af computeren. Men prøv med 130 – 1.

Maskinen påstår hårdnakket at det bliver *negativt*. Det der er sket er at den betragter 130 som et relativt tal, nemlig -126 . Og da $-126 - 1 = -127$ er resultatet ganske rigtigt negativt.

Morale: når du tænker i talrækken 0-255 skal du bruge BCC og BCS, ikke BMI og BPL.

HOP

Der er en anden måde at ændre PC-registret på (og få det til at flytte til en anden instruktion): **JuMP**-kommandoen

JMP (opkode 4C)

Den bruges som regel med absolut adressering, dvs. efterfulgt af en 2-bytes adresse. Denne adresse er adressen for den instruktion du gerne vil have udført nu; adressen puffes simpelt hen over i PC. På næste side ser du et eksempel på et program som bruger JMP til at hoppe forbi et uvedkommende område i hukommelsen.

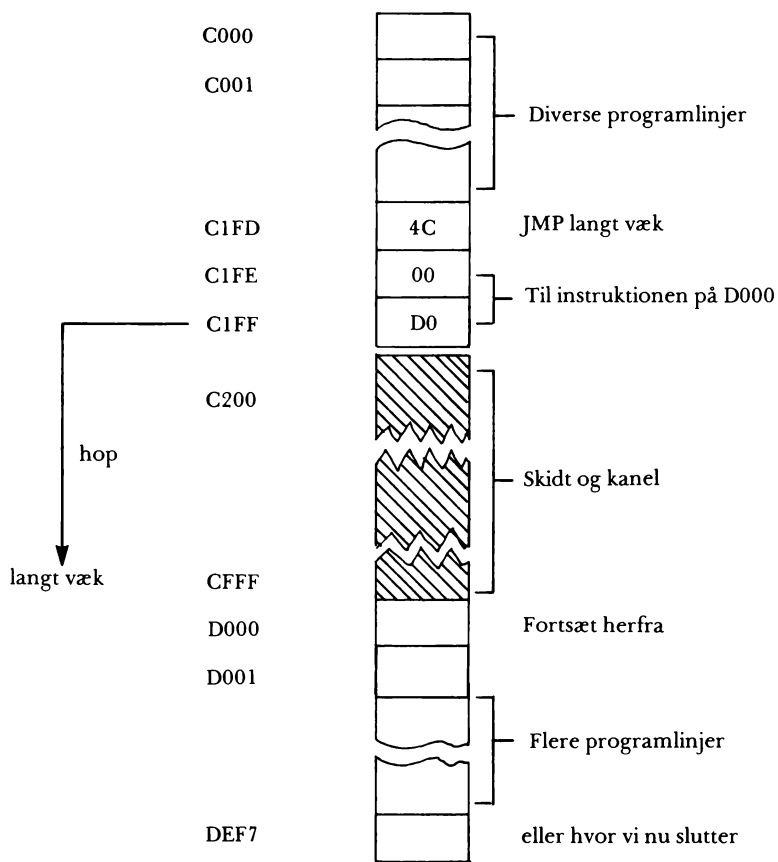
I mnemonic bruges som regel en label:

JMP hop

men man kan også skrive adressen:

JMP D000

Der findes en anden adresseringsmulighed for JMP: *indirekte* adressering. Se kapitel 14.



Med forgreningsinstruktioner kan man skrive en maskinkode der svarer til en FOR-NEXT-løkke i BASIC. Som sædvanlig bliver du nødt til at tænke det grundigt igennem selv.

12 · LØKKER

Lad os tænke os et program som du vil have til at udføre en given opgave flere gange og så stoppe. I BASIC ville du bruge en FOR-NEXT-løkke; i maskinkode må du selv bygge én. Hvis du nogensinde har prøvet at lave en GOTO-version af en FOR-NEXT-løkke (sådan at du kan hoppe ud af den uden at have uvedkommende brokker liggende rundt omkring i maskinen) vil du have forstået princippet allerede. Her ser du to former for BASIC-løkker:

10 FOR K=1 TO 7	10 K=1
20 PRINT "DAVS"	20 PRINT "DAVS"
30 NEXT K	30 K=K+1
	40 IF K<=7 THEN 20

I den anden version bruger vi K som *gennemløbstæller*. Hver gang løkken er gennemløbet forhøjes K med 1, indtil testen i linje 40 til sidst giver negativt udfald og vi forlader løkken. Og det er præcis det samme du skal lave i maskinkode.

Et godt eksempel (på vores nuværende lærdomsstade) er et program som ganger to 8-bits tal med hinanden ved hjælp af gentagen addition og giver et 16-bits resultat. For eksempel

$$17 \times 6 = \underbrace{17 + 17 + 17 + 17 + 17 + 17}_{6 \text{ gange}} = 102 \text{ (decimal)}$$

6 gange

Det er *ikke* den fikseste måde at løse opgaven på, men det er udmærket til at illustrere hvordan en løkke skal konstrueres. Der findes nogle smarte måder der kan pynte lidt på det, men lige i øjeblikket vil jeg hellere lade som ingenting.

De to tal der skal ganges, gemmes i C000 og C001. Det andet tal må ikke være nul. Facit vil gå til C002-C003 som et 2-bytes tal (lille byte/store byte). Jeg får brug for C004 som gennemløbstæller, og C002-C003 kan opbevare mellemresultaterne indtil facit foreligger.

Det giver altså 5 databytes. Her er programmet:

	LDA #00	A9 00	}	Sæt løbende total til nul
	STA C002	8D 02 C0		
	STA C003	8D 03 C0		
	STA C004	8D 04 C0		Sæt tæller på nul
	INC C001	EE 01 C0		Mindre justering for
loop:	CLC	18		korrekt antal løkker
	LDA C002	AD 02 C0		
	ADC C000	6D 00 C0		
	STA C002	8D 02 C0		Lille byte i løbende total
	LDA C003	AD 03 C0		
	ADC #00	69 00		Store byte skal måske have
	STA C003	8D 03 C0		en mente lagt til
	INC C004	EE 04 C0		Forhøj tæller (læg 1 til den)
	CLC	18		
	LDA C001	AD 01 C0		Hent andet tal ind i akkumulatoren
	SBC C004	ED 04 C0		Se om det er lig med tælleren
	BNE loop	D0 <u>E2</u>		Hvis ikke: forgrening tilbage
	RTS	60		Ellers forlad løkken

Det baglæns relative hop på -30 fra BNE loop er understreget (når der regnes med fortegn er -30 lig med E2 hex; se tillæg 1).

INKREMENTERING OG DEKREMENTERING

Du vil have opdaget en ny instruktion

INC (forhøj)

INCrement forhøjer (inkrementerer) indholdet i en lager-celle med 1. Opkoderne er

EE (absolut, ikke-zero-page)

E6 (zero-page)

Desuden findes der to indekserede versioner (se kapitel 13). Tilsvarende har vi kommandoen

DEC (sæk)

DECrement sænker (dekrementerer) med 1 og har opkoderne

CE (absolut, ikke-zero-page)

C6 (zero-page)

Begge kommandoer 'tipper over' uden at tage hensyn til mente (men visse flag påvirkes), sådan at

$$255 + 1 = 0 \quad 0 - 1 = 255$$

hvad disse operationer angår.

SAMMENLIGNING

I eksemplet ovenfor fastlagde vi hvornår løkken skulle være slut ved at trække tællerens tal fra det ønskede antal gen-

nemløb og bruge resultatet til at kontrollere en forgrening. Det er ikke strengt nødvendigt. Et eller andet lyst hoved har tænkt på, at det man tit har brug for er at tage en beslutning på baggrund af *hvad flagene ville have vist* hvis man havde foretaget subtraktionen – men altså *uden* at foretage den. Så var det han opfandt sammenlignings-kommandoen **CoMP**-are:

CMP (opkoder C9, CD, C5 for hhv. umiddelbar, absolut og zeropage-adressering)

Denne kommando justerer flagene nøjagtig som om der var brugt SBC, men uden at ødelægge akkumulatorens indhold. Det er en vældig god idé; man behøver kun flagene for at træffe afgørelsen, og tit er man helst fri for at spolere akkumulatorens indhold.

I rutinen ovenfor kan vi for eksempel erstatte instruktionen SBC C004 med

CMP C004 CD 04 C0

I dette tilfælde gør det faktisk ikke koden kortere, men vi kommer senere til at se eksempler hvor det ganske bestemt gør. Under alle omstændigheder er det en mere kultiveret måde at gå til opgaven på, og i forbindelse med indeksering (kapitel 13) fører det til mere ydedygtige programmer.

INDEKSREGISTRENE

Nu vi alligevel taler om indeksering . . . Jeg har endnu ikke fortalt dig hvad indeksregistrene bruges til. Når de ikke bruges til mere sofistikerede formål (kapitel 13 igen) er de fremragende som gennemløbstællere. De ligger allerede derinde i 6510-chippen, så der er ikke nogen grund til at fumle

med LDA og STA og deslige. Og der er en hærs-kare af instruktioner der kan påvirke dem direkte, så man i en masse tilfælde kan gå helt uden om akkumulatoren.

Som jeg har nævnt er der to indeksregistre: X og Y. Begge er 1-bytes registre. De pågældende kommandoer optræder parvis for X og Y. I tillæg 4 kan du finde de forskellige opkoder og adresseringsmåder. Her vil jeg kort gennemgå instruktionerne (som meget ligner dem vi har set før, men blot bruger X- eller Y-registret i stedet for akkumulatoren):

LDX, LDY	(Loa D X, Y)	indlæs i X eller Y
INX, INY	(INcrease X, Y)	forhøj X eller Y
DEX, DEY	(DEcrease X, Y)	sænk X eller Y
STX, STY	(STore X, Y)	gem indholdet af X eller Y i hukommelsen
CPX, CPY	(ComPare X, Y)	sammenlign X- eller Y-registret med en angiven byte

Man kan forestille sig det sådan: træk den angivne byte fra X-registrets (eller Y-registrets) indhold, justér flagene tilsvarende og glem så resultatet af subtraktionen.

Kommandoerne der forhøjer, sænker og sammenligner er det der gør disse registre til fremragende gennemløbstællere. Lad os skrive ovenstående program om med X-registret som tæller.

Denne gang har vi kun brug for fire databytes: C000 og C001 til de to tal M og N der skal ganges, og C002-C003 til at opbevare lille byte og store byte af resultatet. Så ser det sådan ud:

LDA #00	A9 00	Initialisering
STA C002	8D 02 C0	
STA C003	8D 03 C0	
LDX C001	AE 01 C0	Læg N ind i indeks X
loop: CLC	18	
LDA C002	AD 02 C0	
ADC C000	6D 00 C0	
STA C002	8D 02 C0	
LDA C003	AD 03 C0	
ADC #00	69 00	
STA C003	8D 03 C0	

Så vidt ikke de store forskelle. Men nu kommer pointen mere pludseligt:

DEX	CA	Sænk indeks
CPX #00	E0 00	Sammenlign med nul
BNE loop	D0 <u>E9</u>	Forgrening hvis ikke lig med nul
RTS	60	

E9 i BNE er offset: det er faktisk -23.

Hvis du skal arbejde dig systematisk gennem en blok i hukommelsen eller gennemløbe en række adresser skiftevis, så er her lige hvad du ønsker dig:

13 · INDEKSERING

Måske vil du gerne kopiere en del af hukommelsen over i en anden del. Det kan for eksempel være du vil flytte et maskinkodeprogram fra vores standard-udgangspunkt C000 til et andet sted, eller du vil have et færdigfremstillet billede frem på skærmen og bevare det gamle skærbillede. Med et konkret eksempel: hvis du vil kopiere indholdet af side CE til side CF, skal du starte ved CE00, læse den ind i akkumulatoren, gemme den i CF00, gentage det med CE01 og CF01 osv.

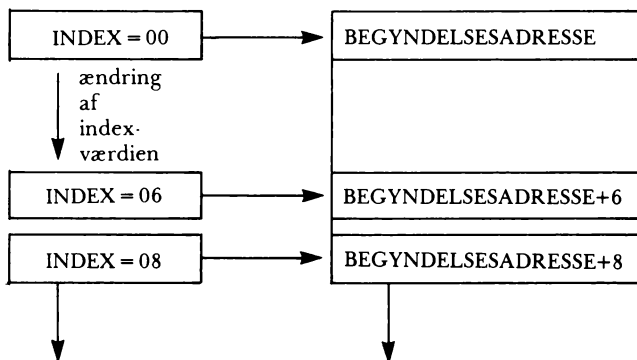
Det er jo herlig ensformigt, og derfor lige noget en computer gerne skulle kunne gøre baglæns, klokken 5 om morgenen, mens den står på hovedet og fløjter “Den gang jeg drog af sted”. Med vores hidtidige udvalg af instruktioner er det desværre ikke tilfældet. På nuværende tidspunkt har vi ikke nogen mulighed for at ændre adressedelen af en opkode.

Det er selvfølgelig ikke helt rigtigt. Vi *kunne* simpelt hen læse en ny byte ind i hukommelsen på det rigtige sted i programmet, ‘omredigere’ opkodens adressebyte mens programmet kører. Det hedder et *selvændrende* program. Dem ser man ikke så tit, og det er godt det samme, for det er svært at sætte fingeren på et program der ikke vil sidde stille (jeg har det samme problem med myg). Er der en fejl i programmet er den ikke til at finde i en programlistning, hvis programmet ændrer i listningen mens det kører. Den lille

synder der udsletter sig selv er en stor frygtet forbryder, som selv en Sherlock Holmes ville ryste i bukserne for.

INDEKSERET ADRESSERING

Med *indekseret adressering* ændrer man ikke reelt opkodens adressebyte, men ændrer dens betydning. Princippet er at den bruges til at angive en *begyndelsesadresse* for en kodeblok, mens et *indeks* fortæller hvor meget længere CPU'en skal gå for at nå til den adresse vi netop gerne vil have fat i. Også det er et 1-bytes offset; men denne gang er det i område 0 til 255 og ikke -128 til 127 som ved forgrening. Og offset-værdien ligger ikke i opkoden, men i det ene af de to registre X og Y. Det giver dig nøjagtig en side at boltre dig på med adresser (selv om man kan krydse sidegrænserne uden problemer – bortset fra en ubetydelig nedsættelse af hastigheden, som kun kan have interesse for en elektronikingeniør). Her er der et grundrids:



Der er fire varianter (som jeg for denne bogs skyld betragter som selvstændige adresseringsmåder, ellers ville tillæg 3 blive alt for forvirrende), nemlig alle kombinationer af X-regi-ster og Y-register med zeropage og ikke-zeropage som be-

gyndelsesadresse. Opkoderne findes i tillæg 4. I virkeligheden har vi ikke meget fornøjelse af en begyndelsesadresse på side nul; det ville kun give os adgang til side 0 og 1, der som allerede nævnt er blevet hugget af de herrer som konstruerede operativsystemet (for at det ikke skal være løgn er side 1 endda *stakken* som SP-registret henviser til). De eneste adresseringsmåder der virkelig kan interessere os er derfor de to ikke-zeropage-adresseringer: én for X og én for Y. I denne henseende er der ingen særlig forskel på X og Y; jeg vil for det meste bruge X.

OVERFØRSEL AF EN SIDE I HUKOMMELEN

Til en begyndelse er der her et program som løser den opgave vi begyndte med – at overføre side CE til CF:

```

        LDX #00          A2 00
loop:   LDA CE00,X       BD 00 CE
        STA CF00,X       9D 00 CF
        INX              E8
        CPX #00          E0 00
        BNE loop         D0 F5      (offset -11)
        RTS              60

```

Bemærk at mnemonics med indekseret adressering har dette format:

LDA (to adressebytes),X

når vi bruger X-registret som indeks (ved zeropage bruges samme format, men uden begyndelsesadressens sidetal 00).

Indekset virker som følger. Kommandoen

```
LDA CE00,X
```


giver computeren meddelelsen 'læg indholdet af X til CE00 og lad akkumulatoren fra den adresse'. Når X tæller 0, 1, 2, ... bliver akkumulatoren efterhånden ladet med alle bytes for side CE. Tilsvarende betyder

```
STA CF00,X
```

'læg indholdet af X til CF00 og gem akkumulatorindholdet i den adresse'.

Al indeksering arbejder på denne måde: læg indekset til begyndelsesadressen for at finde den effektive adresse som skal benyttes.

Der er en anden vigtig ting man skal lægge mærke til: brugen af CPX #00 til at teste om løkken er slut. Det vigtige er at vi forhøjer X før testen. Ved første gennemløb indeholder X altså 1 på det tidspunkt hvor testen sker; derefter indeholder X-registret 2, 3, ..., 255. Ved det sidste (256.) gennemløb vil INX skubbe det op på 256, men dette tal tipper rundt og giver 0 (carry-bitten falder fra), og løkken slutter altså på det sted. Til tider er det faktisk ganske nyttigt at 1-bytes addition tipper over fra 255 til 0 – bare man ikke glemmer det, ellers kan man blive meget overrasket over hvad der sker i programmet!

Læg rutinen ind med LOADER for at teste den, og tryk S til slut. Skriv så følgende BASIC-program:

```
5000 FOR T=0 TO 255
5010 POKE 52736+T,T:POKE 52992+T,119
5020 NEXT
```

Programmet fylder side CE med 0, 1, 2, ..., 255 i rækkefølge, og hele side CF med 119 (77 hex). (Nedenfor kan du finde en maskinkode-måde at gøre det på). Så starter vi maskinkoden:

```
SYS(49152)
```

og kigger igen på side CF:

```
5040 FOR T=0 TO 255
5050 PRINT PEEK(52992+T),
5060 NEXT
```

Start med GOTO 5000. Læg mærke til, og husk det til senere sammenligning, hvor lang tid det tager for BASIC at udføre linje 5000-5020. Det er *langsomt*. Du ser at side DF har skiftet indhold til 0, 1, 2, . . . , 255, ganske som jeg påstod.

Bemærk også at du ikke behøver starte indekseringen ved begyndelsen af siden. Begyndelsesadressen kan være en hvilken som helst (2-bytes) adresse, og den blok du arbejder med kan altså ligge tværs over grænsen mellem to på hinanden følgende sider.

UDFYLDNING AF EN SIDE MED DATA

Ovenfor brugte jeg BASIC til at POKE ting ind på side CE og CF (og gjorde opmærksom på hvor langsomt det er: omkring 5-10 sekunder tager det faktisk). Det er fordi jeg regnede med du ville have mere tillid til BASIC end til maskinkode. Når man tester noget er det klogt kun at bruge midler som man *ved* virker rigtigt. Men med maskinkode kan man let og hurtigt sætte data op på siderne CE og CF.

For at fylde side CF med 77-tal (hex) skriver vi:

```
LDX #00      A2 00
LDA #77      A9 77
loop: STA CF00,X  9D 00 CF
INX          E8
CPX #00      E0 00
BNE loop     D0 F8
RTS          60
```

Med en variant kan vi fylde side CE med 0, 1, 2, . . . , 255:

LDX #00	A2 00
loop: TXA	8A
STA CE00,X	9D 00 CE
INX	E8
CPX #00	E0 00
BNE loop	D0 <u>F7</u>
RTS	60

Læg mærke til den nye kommando

TXA (opkode 8A) **T**ransfer **X**-register to **A**ccumulator

‘overfør X-register til akkumulator’, som kopierer X til A (men uden at ødelægge X). Der er flere lignende kommandoer, som angår X-, Y- og A-registrene:

TAX (opkode AA) **T**ransfer **A**ccumulator to **X**-register

TYA (opkode 98) **T**ransfer **Y**-register to **A**ccumulator

TAY (opkode A8) **T**ransfer **A**ccumulator to **Y**-register

Det er nærliggende at tro at man kan erstatte

```
TXA
STA CE00,X
```

med

```
STX CE00,X
```

men *den går ikke*. Man kan ikke bruge X-indeksering til at lave numre med X selv (tanken om et X-register der æder sin egen hale er så uhyggelig at 6510 nægter at tænke den).

FLERE SIDER

Hvis du gerne vil overføre (eller fylde) mere end 256 data-bytes, har du tre muligheder:

1. Snyder. Hvis det bare drejer sig om to-tre sider, så gentag programmet to-tre gange efter hinanden med forskellige sidetal.
2. Skriv et selvændrende program, hvor den byte der indeholder sidetallet ændres med en STA-instruktion. Jeg har ganske vist frarådet selvændrende programmer, men der kan være situationer hvor det lønner sig, selv om det betragtes som dårlig stil.

Lad os sige du vil fylde siderne fra C4 til CF med byteværdien 77 (hvilket er typisk (ikke de 77) for forskellige 'initialiserings'-rutiner, og vi støder på et lignende problem i kapitel 20). Det er tolv sider hukommelse. Her ser du en selvændrende rutine der gør dette, idet den bruger rutinen ovenfor til at fylde siderne, og en anden løkke, kontrolleret af Y-registret, til at håndtere de tolv sider med. Programmet begynder i C000, standardområdet:

LDY #C4	A0 C4	
LDX #00	A2 00	
LDA #77	A9 77	
loop: STA C400,X	9D 00 C4	sidetal
INX	E8	
CPX #00	E0 00	
BNE loop	D0 <u>F8</u>	
INY	C8	
STY sidetal	8C <u>08</u> <u>C0</u>	(se nedenfor)
CPY #D0	C0 D0	
BNE loop	D0 <u>F0</u>	
RTS	60	

Når vi tæller os frem fra C000 kan vi konstatere at byten for det sidetal som vi vil ændre er lagret på adresse C008. Derfor de dobbelt understregede bytes i opkoden. De to BNE-offsetværdier er henholdsvis -8 og -16.

Hvis du er med så langt vil du have let nok ved at starte og gennemprøve ovenstående (lav et passende BASIC-program til at kigge i lagerområdet mellem side C4 og CF, svarende til 50176-53247 i decimal). Så det vil jeg overlade dig som en øvelse.

Det med de selvændrende programmer vil jeg godt slå helt fast: prøv engang at ændre rutinens første linje til det forkerte

```
LDY #BF      AO BF
```

og lad som om du ikke har opdaget nogen fejl.

Så start programmet. Det virker selvfølgelig ikke. Derfor vil du liste dit program for at underkaste det en nærmere undersøgelse. Hvad er der sket med den skyldige byte?

Ikke nok med at du har vanskeligheder med at afbryde programmet når det kører fast; du vil også opdage at fejlen har overskrevet *sit eget område* med 77!

3. Den tredje (og allerbedste) måde er 'indekseret indirekte adressering'. Det handler næste kapitel om.

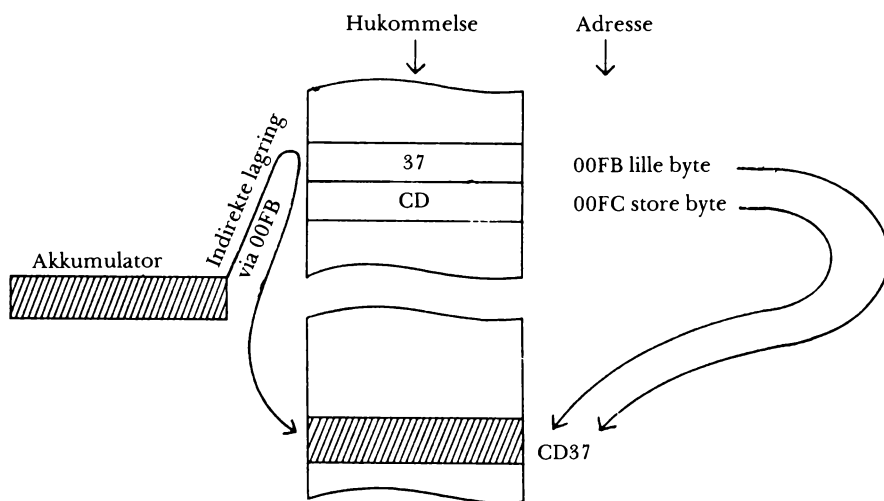
Med den sidste gruppe adresseringsmåder kan du lægge denne besked på en adresse: 'denne adresse må ikke benyttes, kun den adresse som jeg nu opgiver dig'. Det er skattejagtsprincippet overført på adressering.

14 · INDIREKTE ADRESSERING

Under en virkelig skattejagt må du følge en række spor. Hvert spor henviser dig til det næste spor. Indirekte adressering er nogenlunde det samme, blot udføres processen kun én gang. Når du angiver en adresse i opkoden er det *ikke* den adresse der skal bruges i en efterfølgende operation, men en *henvisning* til hvor den adresse skal findes.

Lad os for eksempel forestille os at adresserne 00FB og 00FC indeholder henholdsvis byteværdierne 37 og CD. Instruksen 'gem akkumulatorindholdet indirekte med 00FB' ville have samme virkning som 'gem akkumulatorindholdet i adresse CD37'. Adressen 00FB i opkoden indeholder lille byte for den aktuelle adresse, og den næste byte 00FC indeholder store byte for den aktuelle adresse. Sådan som det er vist på næste side.

Hvorfor skal man igennem alt det besvær, når man udmærket kunne lagre akkumulatorindholdet direkte i CD37? Svaret er at vi let kan tilføje nye instruktioner til programmet der ændrer indholdet i 00FB-00FC, og dermed ændre den adresse som vi gemmer ting i. Men vi kan gøre det uden de farer som lurer i et selvændrende program, for 00FB-00FC er en del af dataområdet, ikke af programområdet (ganske vist ikke det område som vi plejer at lægge data i, men det er der en god grund til; se nedenfor). Derfor kan vi inden

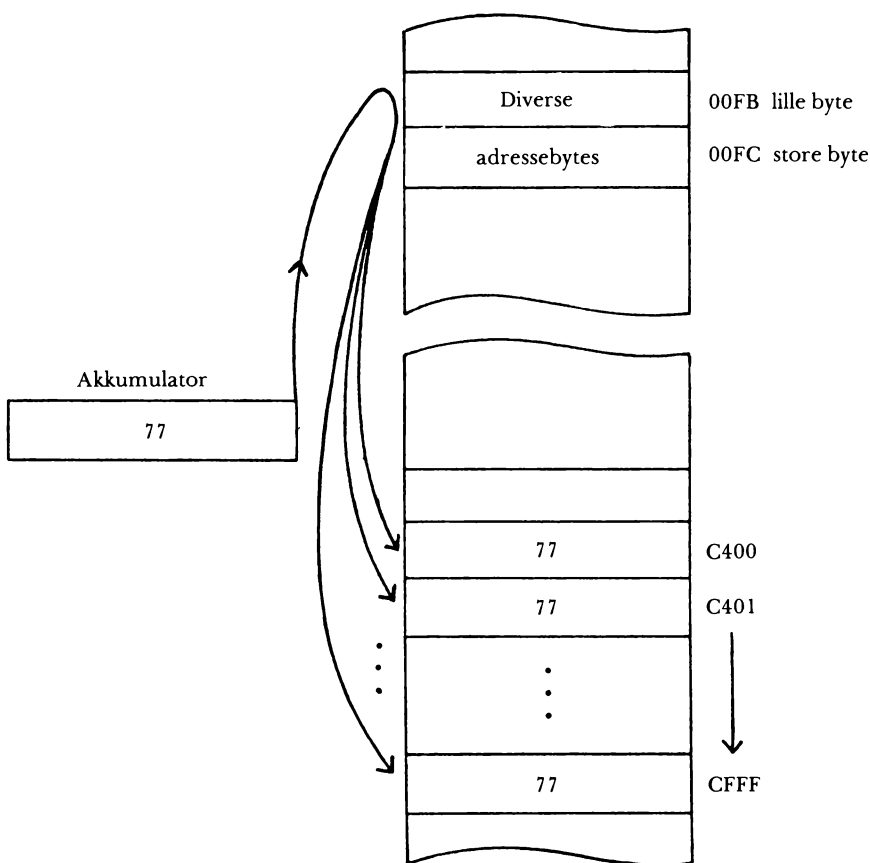


for programmet omdirigere akkumulatoren til et hvilket som helst sted vi ønsker inden for samtlige 64K RAM.

Hvis du synes det er lidt kompliceret, vil du sikkert blive glad for at høre at det er den *enkle* udgave af hvad der sker. Der findes også en indekseret version – ja indeksering kan faktisk ske på to forskellige måder! Men til en begyndelse vil jeg se bort fra indeksets rolle ved at sætte det pågældende register til nul. Først når vi har set hvad en rigtig vaskeægte indirekte adressering kan, vil jeg komme ind på indekseringen også.

UDFYLDNING AF FLERE SIDER MED DATA

Lad os vende tilbage til problemet med at fylde siderne C4 til CF med byteværdi 77. Princippet er at indlæse 77 i akkumulatoren og lagre denne værdi indirekte ved hjælp 00FB-00FC. En løkke vil få indholdet i disse to bytes til at gennemløbe de ønskede adresser C400-CFFF efter tur.



Det er ikke tilfældigt at jeg har brugt zeropageadresserne 00FB-00FC til *adressehenvisningen*. Det blev jeg nødt til. Indirekte adressering er kun mulig med en adressehenvisning på zeropage. De bytes der henviser til den effektive adresse *skal* stå på side nul (det betyder *ikke* at selve den effektive adresse skal stå på side nul, men kun 'sporet' som oplyser hvor adressen findes). Som vi har set tidligere betyder det, at de eneste sikre steder for adressehenvisninger er 00FB-00FE, som operativsystemets skabere så venligt har skænket os (tak til de herrer og/eller damer – vi påskønner *virkelig* den smukke tanke).

Her er koden:

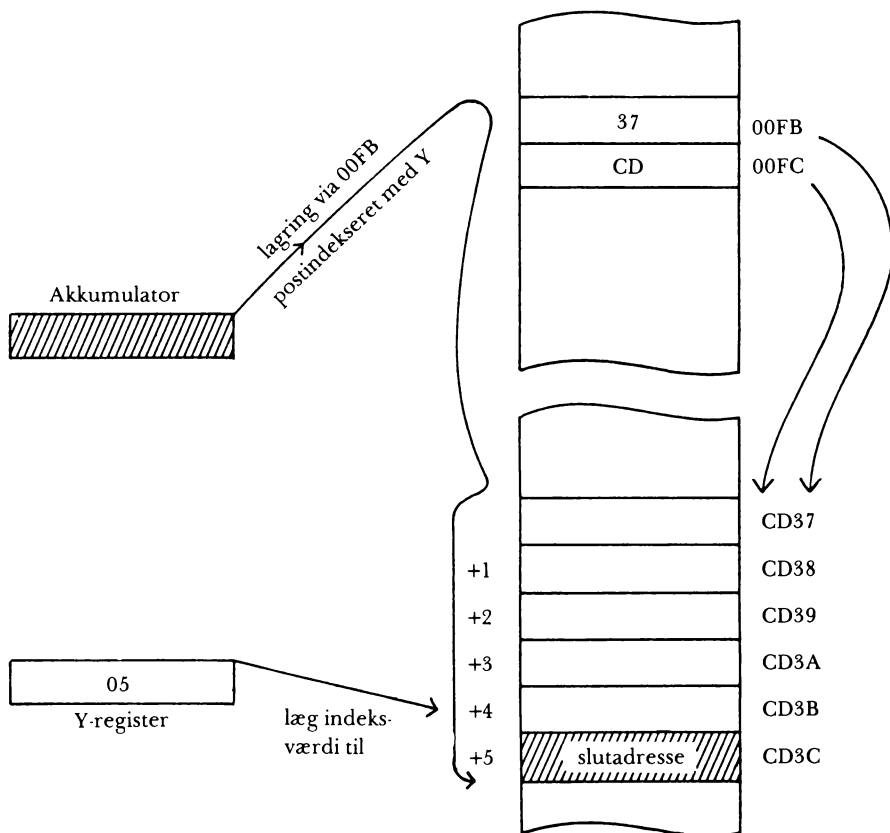
	LDA #00	A9 00	} sæt indirekte adresse til begyndelsen af området
	STA FB	85 FB	
	LDA #C4	A9 C4	
	STA FC	85 FC	
	LDY #00	A0 00	(se bort fra Y-registret)
loop:	LDA #77	A9 77	} indirekte adressering med 00FB (se bort fra Y's rolle her)
	STA (FB),Y	91 FB	
	CLC	18	} forhøj 2-bytes adressehenvisning
	INC FB	E6 FB	
	LDA FB	A5 FB	
	CMP #00	C9 00	
	BNE spring	D0 <u>02</u>	
	INC FC	E6 FC	
spring:	LDA #00	A9 00	} test lille byte
	CMP FB	C5 FB	
	BNE loop	D0 <u>EB</u>	
	LDA #D0	A9 D0	} test store byte
	CMP FC	C5 FC	
	BNE loop	D0 <u>E5</u>	
	RTS	60	

Læg mærke til at vi bruger adresse D000, ikke CFFF, til at teste om programmet er slut, fordi adressen er blevet forhøjet *før* testen foregår. CFFF gennemløber altså løkken og forhøjes til D000, hvorefter løkken ender. Det er meget let at komme til at gennemløbe løkken en gang for lidt eller en gang for meget, så det vil være klogt at undersøge det hvis du kan, og finde ud af præcis hvad der sker i begyndelsen og slutningen.

POSTINDEKSERET INDIREKTE ADRESSERING

Y'et efter mnemonic STA (FB),Y ovenfor betyder at vi her bruger hvad der somme tider kaldes *postindekseret indirekte*

adressering. 'Post' betyder 'efter' og hentyder til at Y-registrets indhold lægges til *efter* at henvisningen til slutadressen er fundet på 00FB-00FC. Hvis vi tager adresserne ovenfor og har 05 i Y-registret ser det hele sådan ud:

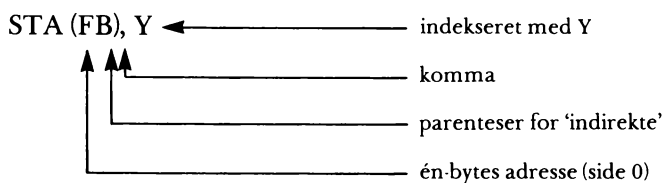


Ved at udnytte dette kan vi få et mere effektivt program: lade 00FB være nul og i stedet forhøje Y-registret med én ad gangen mens en side gennemløbes; så først forhøje sidetal-let i 00FC. Dermed får vi denne kode:

	LDA #00	A9 00
	STA FB	85 FB
	LDA #C4	A9 C4
	STA FC	85 FC
loop1:	LDA #77	A9 77
	LDY #00	A0 00
loop2:	STA (FB),Y	91 FB
	INY	C8
	CPY #00	C0 00
	BNE loop2	D0 <u>F9</u>
	INC FC	E6 FC
	LDA FC	A5 FC
	CMP #D0	C9 D0
	BNE loop1	D0 <u>ED</u>
	RTS	60

Det blev ti bytes mindre, en reduktion på 25 % af længden.

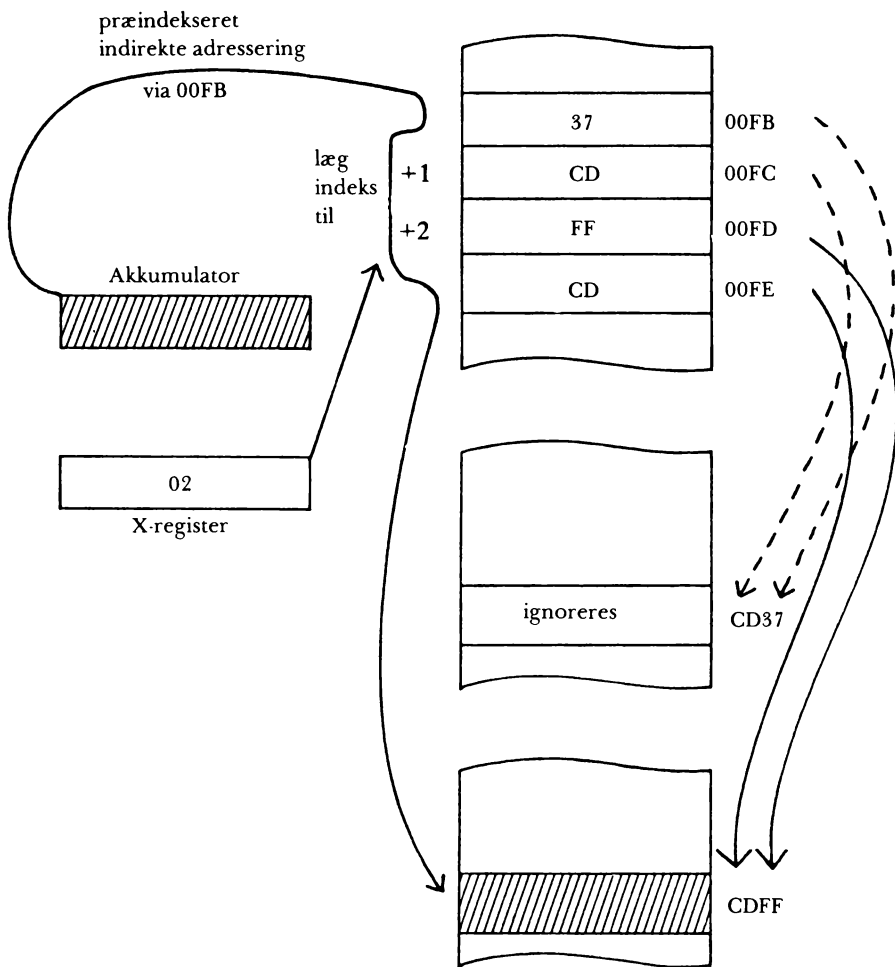
Bemærk formatet for instruktionen:



Modsætningen til postindekseret indirekte adressering hedder

PRÆINDEKSERET INDIREKTE ADRESSERING

Denne adresseringsmåde bruger X-registret. 'Præ' betyder 'før' og her lægges indekset til adressen *før* den effektive



adresse findes. Det vil sige: X-registrets værdi lægges først til adressehenvisningen på 00FB-00FC, og denne værdi udgør så henvisningen til den effektive adresse. Hvis X-registret forhøjes med én, står adressehenvisningen også én position senere. Formatet for instruktionen minder os om at regi-

stret er X og at det lægges til adressehenvisningen, for eksempel:

STA (FB,X)

Jeg kunne godt bryde ud i sang og dans af glæde over den præindekserede adressering (fordi det er en fiks idé der kan bruges til mange smarte ting). Men . . . den sædvanlige hindring, bare meget værre denne gang; selve adressehenvisningen vil jo flytte rundt på side 0, og derfor kunne vi bruge masser af ekstra plads til at lægge adressehenvisninger på netop på side 0. Det har vi bare ikke, vi har sølle fire bytes. Så det eneste tilfælde hvor det kan betale sig at bruge præindekseret adressering er hvis vi bruger indirekte adressering med én adresse (sætter indeks på 0 og lader det blive dér), eller hvis vi vil springe frem og tilbage mellem to alternative muligheder (flytte indeksværdien fra 00 til 02 og tilbage igen – 02 fordi adresserne er to bytes lange). Som følge deraf vil jeg ikke sige mere om det.

INDIREKTE ADRESSERING

Den sidste indirekte adresseringsmåde findes *kun* ved JMP-kommandoen; det er ren og skær indirekte adressering uden avanceret brug af indekser. For eksempel betyder

JMP (CODE)

'kig på adresse CODE og betragt den som lille byte i en to-bytes adresse; brug den næste byte C0DF som store byte; hop til den deraf fremkomne adresse'.

Det er maskinkodens svar på BASICs ON . . . GOTO, for dem der kender den. Princippet er at der opbevares en liste over adresser som man eventuelt kunne ønske sig at hoppe

til. Prop den rigtige værdi ind i C0DE-C0DF, hvor den vil blive brugt til at styre hoppet. Jeg giver ikke noget eksempel her, men tænk for eksempel over en rutine hvor der skal forgrenes på syv forskellige måder afhængigt af om adresse C000 indeholder tallet 1, 2, 3, 4, 5, 6 eller 7, og hvor de følgende 14 bytes indeholder de aktuelle adresser (parvis som lille byte/store byte).

En underrutine er et programhop med variabel returadresse – den husker hvor den hoppede fra. Enkelthederne kontrolleres af maskinstakken, som også kan bruges til mellemlagring.

15 · STAK OG UNDERRUTINER

I BASIC kan man bruge underrutiner til at strukturere programmer i nemme, overskuelige portioner. Det gør det lettere at skrive programmerne og at finde fejl i dem. Lad os repetere hvordan de virker. Underrutinen kaldes med kommandoen GOSUB efterfulgt af dens BASIC-linjenummer. Virkningen er som ved GOTO, bortset fra at maskinen 'husker' det linjenummer som den hoppede fra. Ved slutningen af underrutinen giver kommandoen RETURN den besked om at hoppe tilbage til linjen lige efter den den kom fra. På den måde kan den samme underrutine kaldes forskellige steder fra, og alle RETURN vil blive behandlet korrekt. Endvidere kan en underrutine kaldes fra en anden underrutine. Ja faktisk kan en underrutine kalde sig selv, en teknik der kendes som *rekursiv programmering*.

Nogenlunde sådan foregår det også i maskinkode, men med instruktionernes faktiske adresser i stedet for linjenumre (for sådan nogle har de ikke). Analogen til GOSUB er 'hop til underrutine':

JSR (opkode 20) **J**ump to **S**ub**R**outine

som skal efterfølges af en 2-bytes absolut adresse – den adresse der skal hoppes til. Analogen til RETURN er 'retur fra underrutine':

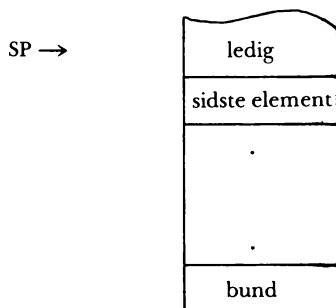
RTS (opkode 60) **Re**Turn from **S**ubroutine

som vi allerede har mødt; det er det obligatoriske 'retur til BASIC' som alle vore maskinkoderutiner ender med (faktisk behandler 64'eren maskinkoderutinen som var den en underrutine under dens enorme BASIC-operativsystem, hvilket er grunden til at vi skal vende tilbage på den måde).

MASKINSTAKKEN

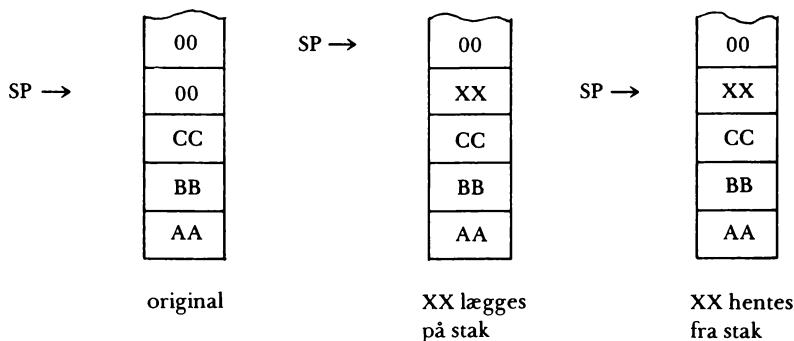
Hvordan virker disse instruktioner? Hop-delen af det fungerer næsten som det almindelige JMP-hop: den nye adresse indlæses i PC-registrets to bytes, hvorved 6510 narres til at vende blikket mod et andet område i programlageret. Men hvis der ikke skete andet ville JSR være det samme som JMP. JSR-instruktionen har også en anden funktion: den gemmer adressen på den instruktion som følger efter JSR, og når maskinkoderutinen er slut ved RTS kan denne adresse findes frem fra lageret og stikkes tilbage i PC, som så kan fortsætte hovedprogrammet hvor den slap. Dette sker ved hjælp af en *stak*.

En stak er en del af hukommelsen med en fast 'bund' og et variabelt 'låg' (på 64'eren ligger maskinstakken altid på side 1). Stakpilen eller SP-registret indeholder adressen for stakkens låg. Pilen, også kaldet pointeren, peger på stakkens åbne top sådan her:



Man kan lægge flere elementer på stakken ved at flytte SP én op og lægge næste byte i hukommelsen, og de kan tages ned fra stakken ved at flytte SP én ned (det er nemlig ikke nødvendigt at slette den byte man tager fra stakken, for stakrutinerne ignorerer alt oven over SP).

For eksempel:



SP peger altså faktisk på den første *ledige* position.

PUSH OG PULL

Selv om JSR-instruktionen klarer alt stakarbejdet for os, er der nogle kommandoer som lader os håndtere stakken direkte. De er også meget nyttige: du kan lægge noget på stakken (push) som du vil gemme midlertidigt, og hente det fra stakken igen (pull) når du får brug for det. Det eneste du skal passe på er at du ikke i mellemtiden har lagt noget andet øverst i stakken! Kommandoerne er

PHA (opkode 48) **PusH** Accumulator to stack

PLA (opkode 68) **PulL** Accumulator from stack

som gemmer og henter akkumulatorindholdet, og

PHP (opkode 08) **P**us**H P**-register to stack
PLP (opkode 28) **P**ul**L P**-register from stack

som gør det samme med alle flagene.

En stak arbejder efter princippet 'sidst ind – først ud'. Forestil dig en bogstabel på et bord. PHA betyder 'læg en bog oven på stablen', og PLA betyder 'tag en bog fra oven'. Sådan bruger du PHA med tre bøger:

PHA 'Robinson Crusoe'
PHA 'Hjortens Flugt'
PHA 'Hobbitten'

og med PLA får du dem fra stablen i denne rækkefølge:

PLA 'Hobbitten'
PLA 'Hjortens Flugt'
PLA 'Robinson Crusoe'

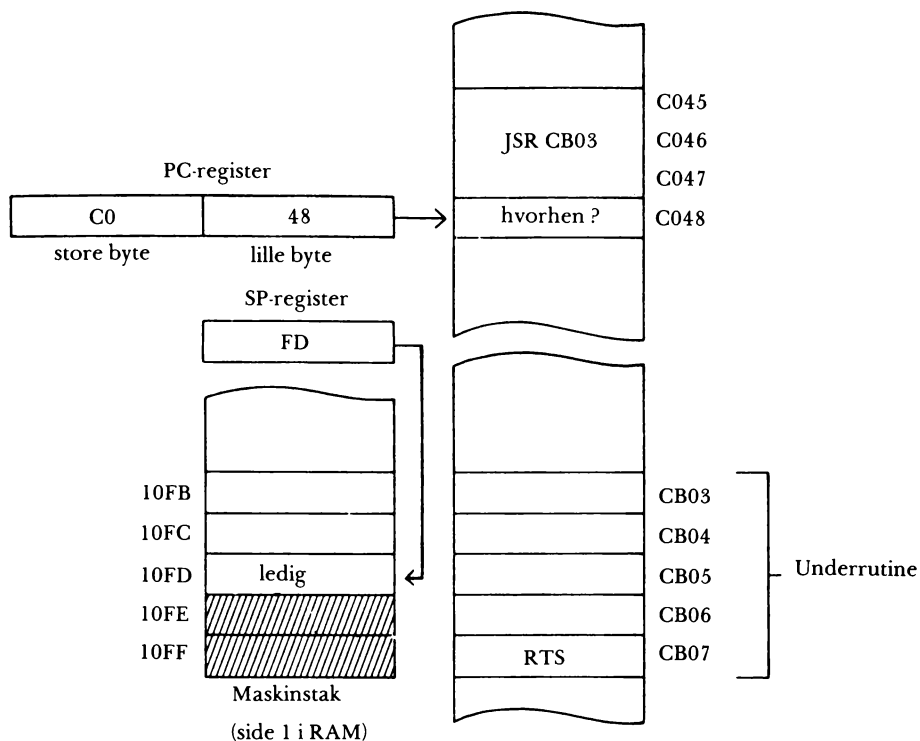
HVORDAN EN UNDERRUTINE BENYTTER STAK

Underrutiner lægger deres returadresser på stakken og henter dem igen når de støder på et RTS. Da adresser optager to bytes, lægges og hentes de i portioner af to bytes (store byte lægges først og hentes sidst, men det får vi nok ikke brug for at vide). Men hvis du har brugt stakken i en under-rutine må du være sikker på at hente alt hvad du har lagt der, inden du vender tilbage; ellers vil du sandsynligvis blive sendt tilbage til en forkert adresse. Det gælder også for det afsluttende 'retur til BASIC': *lad aldrig gammelt ragelse ligge og flyde i stakken.*

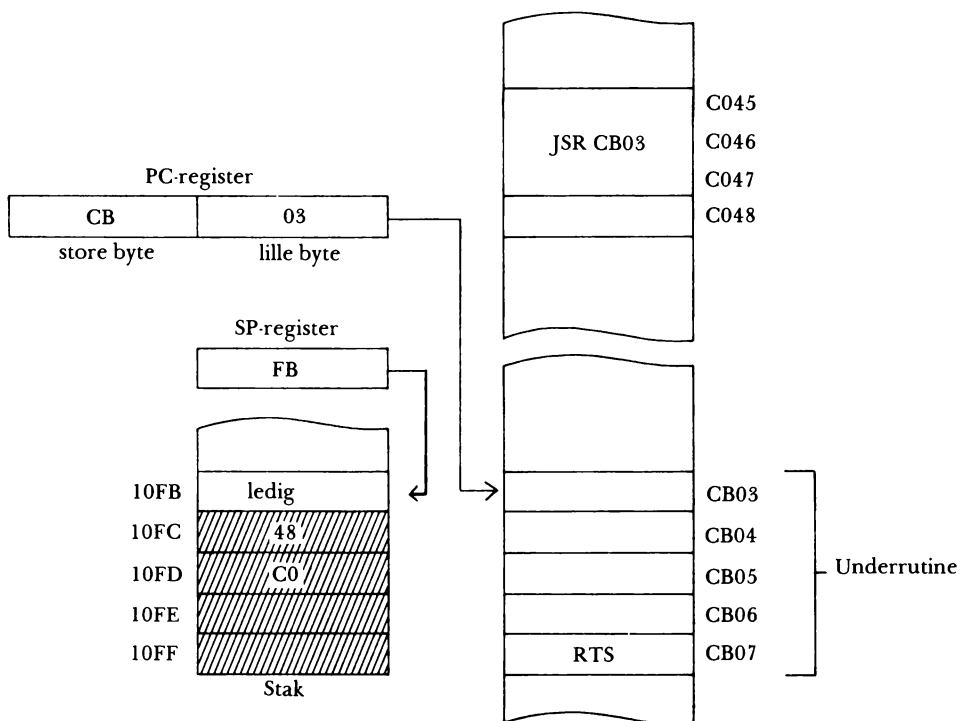
Her et eksempel. CPU'en har netop læst instruktionen

JSR CB03

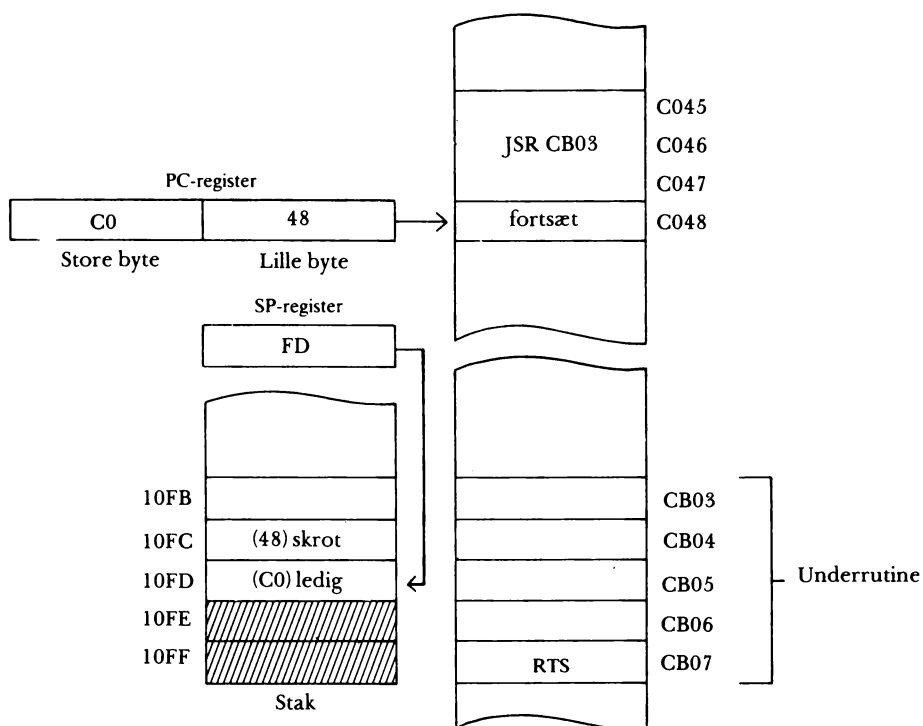
og har rykket PC videre til den næste instruktion på C048:



Nu tager den de to bytes fra PC og lægger dem på stak, flytter SP til den nye 'topadresse' og lægger CB03 ind i PC for at få programmet til at hoppe til underrutinen:



CPU'en arbejder sig så gennem underrutinen indtil den når til RTS. På dette tidspunkt hentes PC fra stak, SP flyttes ned igen, og kontrollen er genetableret inden for hovedprogrammet:



Jeg gentager at *alt dette sker automatisk*. Men når du ved præcis hvad der sker, skulle du have lettere ved at forstå hvordan du kan bruge underrutiner, og hvad der kan gå galt hvis du fuser med stakken.

ET EKSEMPEL

Som eksempel på brugen af underrutiner vil jeg skrive en rutine der gennemløber en side i hukommelsen og erstatter alle bytes som ligger uden for et bestemt område (for eksempel 48-57 decimal, ASCII-koderne for tallene 0-9) med en bestemt byte (for eksempel 32, ASCII for 'mellemrum'). Vi skal bruge fire databytes:

- C000 Sidenummer for proces
- C001 Byte som skal indsættes hvis uden for område
- C002 Områdets nedre grænse
- C003 Områdets øvre grænse plus 1

Underrutinen skal udføre opgaven 'erstat byte med 32'. Det viser sig (når koden er skrevet) at underrutinen vil begynde på adresse C029. Hovedprogrammet på C004.

	LDA C000	AD 00 C0	} Indlæs startadresse til indirekte adressering i 00FB-00FC
	STA FC	85 FC	
	LDA #00	A9 00	
	STA FB	85 FB	
	LDY #00	A0 00	
loop:	SEC	38	
	LDA (FB),Y	B1 FB	} Undersøg om byte er under grænsen
	CMP C002	CD 02 C0	
	BCS spring1	B0 <u>03</u>	
	JSR erstat	20 <u>29 C0</u>	Underrutine: udregn adresse senere
spring1:	SEC	38	
	CMP C003	CD 03 C0	} Undersøg om byte er over grænsen
	BCC spring2	90 <u>03</u>	
	JSR erstat	20 <u>29 C0</u>	Underrutine for anden gang
spring2:	INY	C8	
	CPY #00	C0 00	
	BNE loop	D0 <u>E7</u>	Relativt hop på -25
	RTS	60	Retur til BASIC

Når du skriver dette af ved du ikke hvad adressen for underrutinen vil blive, så du kan ikke udfylde værdierne for JSR. Anbring to understegninger (skriv dem begge to, så er det lettere at tælle offset-værdien for BNE rigtigt). Nu tæller vi op, og vi ser at næste ledige adresse er C029. Sådan skriver vi underrutinen:

erstat: LDA C001	AD 01 C0	
STA (FB),Y	91 FB	Indirekte indksering
RTS	60	Retur til hovedprogram

Skriv først, før du gennemprøver det, en BASIC-rutine som fylder side CF med tilfældige værdier. Indlæs så disse fire databytes:

CF 20 30 3A

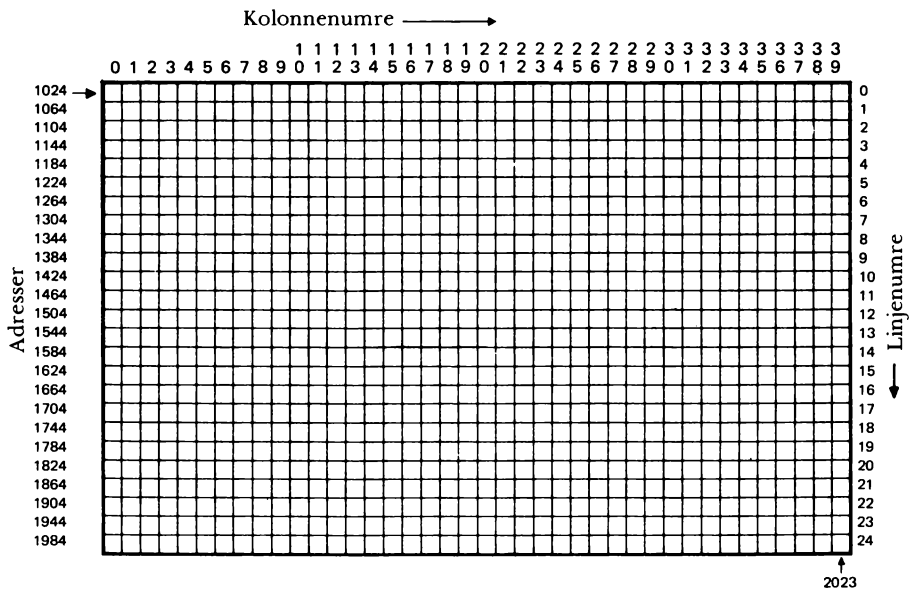
og start det. Undersøg at der kun er værdier tilbage inden for området 48-57; alle andre skal være blevet til 32.

I næste kapitel vil vi møde en mere dramatisk måde at bruge denne rutine på: på skærmen. Der vil du kunne se bytes skifte værdi!

Billedet du ser på din monitor er resultatet af information der er oplagret i to områder af hukommelsen. Ved at ændre indholdet i disse områder kan du lave numre med skærbilledet.

16 · SKÆRM- OG FARVEKONTROL

Skærbilledet består af 25 linjer, hver med 40 kolonner. Linjerne er nummereret fra 0 til 24, kolonnerne fra 0 til 39. Det giver 1000 karakterer i alt:



SKÆRMHUKOMMELSE

Det lagerområde som angiver karaktererne på skærmen kaldes *skærmhukommelsen* eller *Video RAM*, og det går fra

1024 til 2023 decimal (0400-07E7 hex). Eftersom computerens hukommelse ikke har nogen naturlig rektangulær form, ligger alt som en lang række adresser i hukommelsen. Adresserne løber hen ad linjerne og rykker så ned på næste linje og frem til første kolonne når linjen er slut, nøjagtig ligesom man læser i en bog. Hexadresserne for skærmhukommelsen svarer altså til disse positioner på skærmen:

0400	0401	0402	0427
0428	0429	042A	044F
...	...	...	...
07C0	07C1	07C2	07E7

Generelt kan adressen for linje L kolonne K udtrykkes (i decimal):

$$1024 + 40 \times L + K$$

og for at vise en given karakter på en given position behøver vi kun lagre den tilsvarende byte på denne adresse.

Den kode der kræves er *ikke* ASCII, men den kode der er vist i din brugervejlednings tillæg E. Med nogle få undtagelser svarer denne kode til ASCII minus 64 for alfabetets vedkommende, ASCII minus 32 for grafik og god gammeldags ASCII (hvor uopfindsomt!) for tallene 0-9.

OK, lad os kaste os ud i det. For at fremstille en bold i linje 12 kolonne 20 udregner vi først adressen. Den er

$$1024 + 40 \times 12 + 20 = 1524 \quad (05F4 \text{ hex})$$

Koden for en bold er 81 i henhold til tillæg E i brugervejledningen, og det svarer til 51 hex. Vi kan altså bruge denne maskinkoderutine:

LDA #51	A9 51
STA 05F4	8D F4 05
RTS	60

Skriv det ind. Start det ikke, men brug i stedet en BASIC-rutine:

```
5000 PRINT CHR$(147)
5010 SYS(49152)
5020 GOTO 5020
```

Så kan vi begynde med en pæn ren skærm og undgå rodede fejlmeddelelser indtil vi afbryder. Prøv det, og check at det virker. (I nogle tidlige versioner af 64'ers ROM ser det ud som om det ikke virker; men hvis du ændrer baggrundsfarven med POKE 53281,7 vil du få kuglen at se. Den var bare blevet aftegnet i samme farve som baggrunden.)

KARAKTERRÆKKER

Prøv denne rutine på samme måde:

LDA #04	A9 04
STA FC	85 FC
LDA #7B	A9 7B
STA FB	85 FB
LDX #12	A2 12
LDY #00	A0 00
loop: LDA #51	A9 51
STA (FB),Y	91 FB
CLC	18
LDA FB	A5 FB
ADC #28	69 28
STA FB	85 FB

LDA FC	A5 FC
ADC #00	69 00
STA FC	85 FC
DEX	CA
CPX #00	E0 00
BNE loop	D0 <u>EA</u>
RTS	60

Med X som tæller gennemløbes en løkke, og med indirekte adressering lagres byteværdien 51 i en række adresser med en afstand på 28 hex – svarende til 40 decimal. Med andre ord forhøjes linjenummeret mens kolonnennummeret er konstant. Resultatet er en lodret række kugler. Begyndelsesadressen er 047B, hvilket er linje 3 kolonne 3.

Hvis du ændrer ADC # 28 til

ADC #29	69 29
---------	-------

får du en diagonal række, fordi 29 hex er 41 decimal, som giver en forskel på 40 (= 1 linje) plus 1 (= 1 kolonne). Hvis du i stedet prøver

ADC #01	69 01
---------	-------

får du en vandret række. Vil du have en diagonal der går til nederste venstre hjørne tror du måske du skal bruge

ADC #27	69 27
---------	-------

Prøv det. Virker det? Ja-a, lidt. Hvad er det da der er i vejen? At X tipper over fra 0 til 255!

FARVEHUKOMMELSE

Skærmfarverne opbevares i *farvehukommelsen* også kaldet *farve-RAM*. Den er opbygget ganske ligesom skærmhukom-

melsen, men ligger i området 55296 til 56295 decimal (D000-D3FF hex). Farvekoderne er de sædvanlige for Commodore 64:

Sort	00
Hvid	01
Rød	02
Cyan	03
Violet	04
Grøn	05
Blå	06
Gul	07
Orange	08
Brun	09
Lyserød	0A
Mørkegrå	0B
Mellemgrå	0C
Lysegrøn	0D
Lyseblå	0E
Lysegrå	0F

Koderne i farve-RAM'en giver forgrundsfarven (skriftfarven). For at ændre baggrunds- og kantfarve skal man lagre de tilsvarende værdier i adresserne hhv. 53281 og 53280 (D021 og D020).

Rutinen ovenfor kan tilpasses til at ændre farver i stedet for at tegne bolde. Udskift LDA #04 med:

LDA #D8 A9 D8

og LDA #51 med

LDA #07 A9 07

hvis du vil have gule karakterer. Brug dette BASIC-program til at få resultatet at se:

```
5000 PRINT CHR$(147)
5010 FOR T=1 TO 24
5020 PRINT "*****";
5030 NEXT
5040 SYS(49152)
5050 GOTO 5050
```

DA DIN APPETIT BLEV VAKT

Vi kan nu gå tilbage og kigge på den rutine som jeg brugte til at introducere maskinkode med i kapitel 1. Første trin er at *disassemblere* det: oversætte det fra hex til mnemonics. Til dette brug er tillæg 6 meget nyttigt; det giver en liste over alle opkoder i nummerorden med adresseringsmåder.

Resultatet er:

	LDX #00	A2 00
	LDA C0 00	AD 00 C0
gem:	STA 0400,X	9D 00 04
	INX	E8
	CPX #00	E0 00
	BEQ slut	F0 <u>03</u>
	JMP gem	4C <u>06</u> <u>C0</u>
slut:	RTS	60

Der er én databyte i C000 som fra begyndelsen er 00, men som senere ændres med POKEninger.

Det rutinen gør er at fylde side 04 med den byte som er angivet i C000. Nu er 04 begyndelsen af skærmhukommelsen, så du ser en blok af skærmen blive forvandlet til én ka-

rakter. Resten af BASIC-programmet foretager tilfældige ændringer af den pågældende byte hver gang du trykker en tast – og bruger POKE til at skifte sidetal, først gennem skærmhukommelsen, så gennem farvehukommelsen. Bemærk hvor hurtigt dette enkle maskinkodeprogram opnår disse virkninger.

EN 'CIFFERSI'

Nu skal jeg indfri mit løfte fra sidste kapitel og skrive en rutine som benytter skærmen til at vise virkningen af et program som gennemløber en blok af hukommelsen (her skærm-RAM), idet den erstatter alle bytes uden for et udvalgt område med en bestemt byte. Prøv engang at gætte hvad virkningen bliver før du afprøver det!

Der er fire databytes i C000-C003: sidetallet (04), erstatningsbyten (20), nedre grænse for bytes der skal erstattes (30) og øvre grænse (3A). Koden er

```

                LDA C0 00      AD 00 C0
                STA FC        85 FC
                LDA #00       A9 00
                STA FB        85 FB
                LDY #00       A0 00
loop: SEC              38
                LDA (FB),Y    B1 FB
                CMP C002      CD 02 C0
                BCS spring1   B0 03
                JSR erstat    20 31 C0
spring1: SEC          38
                CMP C003      CD 03 C0
                BCC spring2   90 03
                JSR erstat    20 31 C0
spring2: INY          C8

```

```

        CPY #00          C0 00
        BNE loop         D0 E7
        INC FC           E6 FC
        LDA FC           A5 FC
        CMP #08          C9 08
        BNE loop         D0 DF
        RTS              60
erstat: LDA C001         AD 01 C0
        STA (FB),Y       91 FB
        RTS              60

```

Rutinen fungerer lige som eksemplet i slutningen af sidste kapitel, bortset fra at den i stedet for en enkelt side gennemløber siderne 04-07. Det er skærmhukommelsen *plus* et 'uskadeligt' område fra 07E8 til 07FF, hvori datapointerne for sprites ligger. Så der bliver ikke nogen problemer med mindre vi bruger sprites. Hvis vi gør er det nødvendigt at ændre testen for sidste gennemløb, hvilket gør den lidt mere kompliceret (test FB *og* FC).

Rutinen virker på den måde at alle karakterer som ikke er et ciffer 0, 1, . . . , 9 bliver siet fra skærbilledet. Det er fordi den erstatter alle karakterkoder uden for området 48-57 (decimal) med et mellemrum (32 decimal). For at se det i aktion må vi konstruere et interessant skærbillede:

```

5000 PRINT CHR$(147);
5010 FOR T=1 TO 999
5020 PRINT CHR$(40+INT(80*RND(0)));
5030 NEXT
5040 GET A$:IF A$="" THEN 5040
5050 SYS(49156)
5060 GOTO 5060

```

Vent til skærmen er fyldt, og tryk så på en tast. Smask!

En interessant variation er at erstatte linje 5050 og 5060 med

```

5050 FOR K=120 TO 49 STEP -1
5060 POKE 49155,K
5070 SYS(49156)
5080 NEXT
5090 GOTO 5090

```

Endnu en variation er at bruge den seneste version, men tilføje:

```

5005 POKE 49153,83

```

som ændrer én databyte.

OMVENDT SKÆRMFARVE

Hvis man lægger 128 decimal til indholdet af en skærm-adresse skifter den pågældende karakter til 'omvendt skærmfarve' eller 'revers video'; det vil sige at forgrunds- og baggrundsfarven bytter plads. Med en løkke der gennemløber hele skærmhukommelsen kan man lynhurtigt slå hele billedet over på omvendt video:

LDX #04	A2 04	
LDA #04	A9 04	
STA FC	85 FC	
LDA #00	A9 00	
STA FB	85 FB	
LDY #00	A0 00	
loop: CLC	18	
LDA (FB),Y	B1 FB	
ADC #80	69 80	80 hex = 128 decimal
STA (FB),Y	91 FB	
INY	C8	
CPY #00	C0 00	
BNE loop	D0 <u>F4</u>	

INC FC	E6 FC
DEX	CA
SEC	38
CPX #00	E0 00
BNE loop	D0 <u>EC</u>
RTS	60

Hvis det indledende LDX #04 ændres til LDX #01 eller LDX #02 eller LDX #03, vil kun den første, de to eller de tre første sider blive reverse. Her er der en BASIC-rutine der kan illustrere programmets hurtighed:

```

5000 PRINT CHR$(147);
5010 FOR T=1 TO 24
5020 PRINT "11112222333344445555666677778888999
      90000";
5030 NEXT
5040 GET A$:IF A$="" THEN 5040
5050 IF A$="S" THEN STOP
5060 SYS(49152)
5070 GOTO 5040

```

Skærmen fyldes med karakterer, og skærmfarven skifter hver gang du trykker en tast (undtagen S som standser programmet).

PRINT AT

En anden nyttig rutine er 'PRINT AT L,K', som giver mulighed for at PRINTe en given karakter på en given linje og i en given kolonne. Almindelig 64-BASIC mangler denne instruktionsmulighed, men man kan få den samme virkning ved hjælp af cursorkontrollen. Lad os i stedet skrive en maskinkoderutine (faktisk står der allerede én du kan bruge i ROM – se kapitel 21 – men det er lærerigt at skrive sin

egen). Princippet er at udregne $1024 + 40 \times L + K$ og bruge indirekte adressering. '+ K' udføres med indeksering.

Hvordan ganger man med 40 i maskinkode? En løkke der adderer 40 gange kunne klare det, men det er langsomt. I stedet for laver vi tre gange venstreforskydning og har så $8 \times L$. Husk det. To gange venstreforskydning igen giver $32 \times L$. Læg så de $8 \times L$ til, så har du $40 \times L$. Det er jo nemt nok!

Der bliver brugt tre databytes: skærmkode for den karakter der skal PRINTes, linjenummeret og kolonnennummeret. De ligger som sædvanlig fra C000-C002. Jeg foreslår disse data

51 0A 0F

til den første test.

LDA #00	A9 00	}	gem L i FD-FE
STA FE	85 FE		
LDA C001	AD 01 CO		
STA FD	85 FD		
CLC	18	}	gang med 8 'hurtigt og frækt'
ASL FD	06 FD		
ROL FE	26 FE		
ASL FD	06 FD		
ROL FE	26 FE		
ASL FD	06 FD		
ROL FE	26 FE		
LDA FD	A5 FD		
STA FB	85 FB		
CLC	18		
LDA FE	A5 FE		
ADC #04	69 04		
STA FC	85 FC		

CLC	18		
ASL FD	06	FD	
ROL FE	26	FE	
ASL FD	06	FD	
ROL FE	26	FE	
CLC	18		
LDA FB	A5	FB	
ADC FD	65	FD	
STA FB	85	FB	
LDA FC	A5	FC	
ADC FE	65	FE	
STA FC	85	FC	
LDY C002	AC 02	C0	brug indeksering til at lægge K til
LDA C000	AD 00	C0	karakterens skærmkode
STA (FB),Y	91	FB	PRINT karakteren
RTS	60		

gang med 2 to gange til op til $32 \times L$

$8 \times L + 32 \times L = 40 \times L$

Find på en BASIC-rutine der kan teste det grundigt, for eksempel

```

5000 PRINT CHR$(147)
5010 FOR L=0 TO 24
5020 FOR K=0 TO 39
5030 POKE 49153,L:POKE 49154,K
5040 SYS(49155)
5050 NEXT: NEXT

```

som vil checke skærmpositionerne, og med nogle passende POKEninger i 49152 kan du sikre dig at den rigtige karakter bliver PRINTet.

*En kode for den aktuelt nedtrykkede tast er lagret i adresse 197.
Den kan bruges til*

17 · TASTATURKONTROL

Hvis du vil skrive maskinkoderutiner som reagerer på tastetryk (bevægelig grafik for eksempel) må du, inden for maskinprogrammet, have mulighed for at afgøre hvilken tast der bliver trykket på. Det kan gøres ved at kigge i adresse 197 decimal 00C5 hex, som indeholder en (mystisk) kode for den tast som netop er nedtrykket eller kode 64 for 'ingen tast'. Koderne er hverken ASCII eller skærmkoder, og jeg har opført dem i tillæg 8. Prøv at checke dem med et lille BASIC-program:

```
7000 PRINT PEEK(197)
7010 GOTO 7000
```

Start programmet med GOTO 7000, og tryk forskellige taster.

Med en test, der undersøger indholdet af 00C5, og en tilhørende forgrening, kan man opnå tastaturkontrol over sin maskinkode.

LOOP-Y

Her er der et simpelt eksempel. Y-registret kontrollerer en løkke som PRINTer en karakter på skærmen og skriver et mellemrum på begge sider af den. Hvis du ikke trykker på nogen tast flytter karakteren sig støt til højre. Hvis du tryk-

ker 'B' for *baglæns* flytter den til venstre, og hvis du trykker 'S' standser programmet. For enkelhedens skyld flytter karakteren sig gennem én side af skærmhukommelsen.

	LDA #00	A9 00	
	STA FB	85 FB	
	LDA #06	A9 06	
	STA FC	85 FC	
	LDY #00	A0 00	
loop:	LDA #20	A9 20	
	STA (FB),Y	91 FB	
	INY	C8	
	INY	C8	
	STA (FB),Y	91 FB	
	DEY	88	
	LDA #51	A9 51	
	STA (FB),Y	91 FB	
test:	LDA #1C	A9 1C	kode for B
	CMP C5	C5 C5	undersøg om tasten trykkes
	BNE spring	D0 <u>02</u>	
	DEY	88	} Y er allerede rykket en plads til højre; flyt nu to pladser til venstre
	DEY	88	
spring:	LDA #0D	A9 0D	kode for S
	CMP C5	C5 C5	undersøg om tasten trykkes
	BNE loop	D0 <u>E5</u>	
	RTS	60	

Hvis du starter dette vil du opdage at det hele går grassat. Du ser en masse blinkende pletter og inderlig lidt der ligner en vandrende karakter. Grunden er enkel: det går for hurtigt! TV-skærmen kan kun vise 50 billeder i sekundet, og pletten bevæger sig meget hurtigere end som så.

Det er et almindeligt problem i maskinkode; løsningen er at indlægge en forsinkelse. Det gøres lettest med en under-rutine:

TILFØJELSE AF PATCH

Vi kan tilføje en JSR-instruktion, som leder programmet til en 'forsinkelsesrutine'. Det kaldes at tilføje en *patch* til ('sætte en lap på') det originale program.

Vi begynder som før:

```
LDA #00      A9 00
STA FB       85 FB
LDA #06      A9 06
STA FC       85 FC
LDY #00      A0 00
loop: LDA #20  A9 20
      STA (FB),Y  91 FB
      INY          C8
      INY          C8
      STA (FB),Y  91 FB
      DEY          88
      LDA #51     A9 51
      STA (FB),Y  91 FB
```

Her er der et godt sted at lægge patchen:

```
JSR forsink  20 29 C0  udregn hopadressen efter
                                programlistning
```

hvorefter vi genoptager det originale program

```
test: LDA #1C      A9 1C
      CMP C5        C5 C5
      BNE spring    D0 02  offset upåvirket af patch
      DEY           88
      DEY           88
spring: LDA #0D      A9 0D
      CMP C5        C5 C5
      BNE loop       D0 E2  offset ændret pga. patch
      RTS           60
```

Til sidst skal vi tilføje

EN FORSINKELSESLØKKE

Princippet er her at lade X-registret gennemløbe en løkke 256 gange uden at foretage sig noget, hvorefter vi vender tilbage til hovedprogrammet. X-registret er vigtigt i hovedprogrammet, så derfor lægger vi det på stak ved løkkens begyndelse (ved hjælp af akkumulatoren) og henter det igen ved slutningen. Her er koden:

forsink:	TXA	8A
	PHA	48
	LDX #00	A2 00
floop:	DEX	CA
	CPX #00	E0 00
	BNE floop	D0 <u>FB</u>
	PLA	68
	TAX	AA
	RTS	60

Lad være med at slå 'floop' op i ordbogen, det er bare en forkortelse for 'forsinkelsesloop'.

Læg mærke til stakhåndteringen:

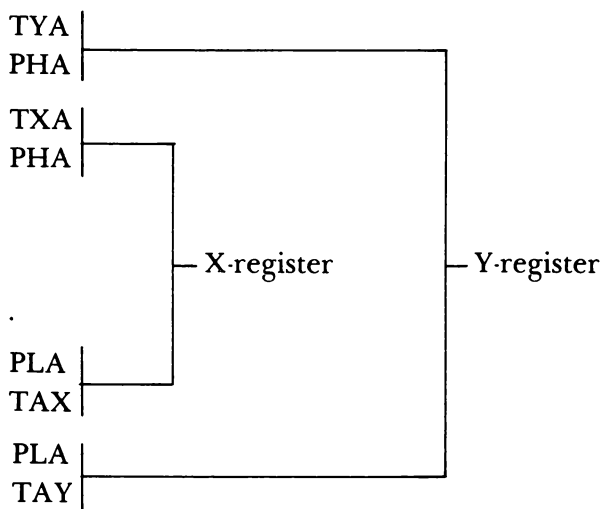
—	TXA	X overføres til A
—	PHA	A (som nu indeholder X) lægges på stak
—	PLA	A hentes fra stak (stadig indeholdende den X-værdi vi vil huske)
—	TAX	A overføres til X (tilbage til udgangspositionen)

Nu tror du måske at en løkke på 256 operationer vil nedsætte hastigheden tilstrækkeligt, men nej! *Det går stadigvæk for hurtigt.* Derfor bruger vi Y-registret til at gennemløbe hele forsinkelsen 256 gange. For 65536 operationer må vel være nok?

Erstat underrutinen ovenfor med den følgende (men uden at røre ved hovedprogrammet):

```
forsink: TYA          98
          PHA          48
          TXA          8A
          PHA          48
          LDY #00      A0 00
          LDX #00      A2 00
floop:   DEX          CA
          CPX #00      E0 00
          BNE floop    D0 FB
          DEY          88
          CPY #00      C0 00
          BNE floop    D0 F6
          PLA          68
          TAX          AA
          PLA          68
          TAY          A8
          RTS          60
```

Læg denne gang mærke til at vi henter X- og Y-registret fra stakken i modsat rækkefølge af hvordan de blev lagt:



Jo, nu går det for langsomt. Levende grafik med snegle fart! Men det kan vi rette på, for nu har vi en prototype på en forsinkelsesløkke, som kan finjusteres ved blot at ændre begyndelsesværdien for Y. Du kan se at det giver et rimeligt resultat hvis LDY #00 (forsinkelse på 256) ændres til

LDY #0A A0 0A (forsinkelse på 8 løkker)

Sæt 0A ned til 05 eller 04 og det går ganske hurtigt, eller sæt det op til 12 eller 16 og det går temmelig langsomt. Når som helst du får brug for en forsinkelse kan du anvende forsinkelsesrutinen med en passende begyndelsesværdi for Y, og senere justere Y-værdien som det passer dig.

Nu vender vi for en kort bemærkning tilbage til programmeringsteorien for at kaste et blik på en anden vigtig gruppe instruktioner:

18 · LOGIK

Der er en sidste gruppe maskinkodekommandoer du bør vide noget om, selv om vi kun får brug for dem i næste kapitel om sprites. Det er de *logiske* instruktioner

AND
ORA
EOR

Først lidt matematisk logik:

ARVEN EFTER GEORGE BOOLE

En matematiker ved navn George Boole fik omkring 1854 den idé at bruge matematiske beregningsmåder til at studere logik med, og han offentliggjorde en bog med titlen *The Laws of Thought*. Han havde ikke en chance for at gætte hvad elektronikingeniører et århundrede senere ville bruge hans ideer til: hans *booleske algebra* er lige sagen ved konstruktionen af computerkredsløb.

Vi kan bruge bittene 0 og 1 til henholdsvis at repræsentere de logiske værdier 'falsk' og 'sand'. Og vi kan foretage udregninger med dem ved hjælp af booleske regler.

Betragt for eksempel sætningen

Det er tirsdag OG det regner.

Hvornår er det sandt? Ville det være sandt hvis det var onsdag? Nej, selv om det så øs-pjask-sjaskregnede. Og hvis det nu *var* tirsdag, men solen skinnede og naboerne lå og dasede i badetøj, så ville det stadig ikke være noget sandt udsagn. *Begge* dele af et OG-udsagn må være sande for at det hele er sandt. Eller som Boole grundlæggende formulerede det (omend med andre symboler):

$0 \text{ AND } 0 = 0$ (falsk OG falsk = falsk)

$0 \text{ AND } 1 = 0$ (falsk OG sandt = falsk)

$1 \text{ AND } 0 = 0$ (sandt OG falsk = falsk)

$1 \text{ AND } 1 = 1$ (sandt OG sandt = sandt)

Du kender nok dette princip fra BASIC, og i maskinkode bruges det på nogenlunde samme måde, som vi får at se.

Der findes også et ELLER-udsagn; på engelsk hedder det OR, på 6510'sk hedder det ORA:

$0 \text{ ORA } 0 = 0$

$0 \text{ ORA } 1 = 1$

$1 \text{ ORA } 0 = 1$

$1 \text{ ORA } 1 = 1$

Det går ud fra den tanke at 'p ELLER q' er sandt hvis en af delene er: 'det sner ELLER jeg er en røget sild'. Vi holder ikke på begge dele!

Endelig er der i dette tankesystem det *eksklusive ELLER*, også kendt som EOR (Exclusive OR). Her er 'p EOR q' ensbetydende med 'p ELLER q men *ikke* begge', og deraf følger

$0 \text{ EOR } 0 = 0$

$0 \text{ EOR } 1 = 1$

$1 \text{ EOR } 0 = 1$

$1 \text{ EOR } 1 = 0$ bemærk forskellen!

BYTELOGIK

Sådan virker de logiske operationer på enkelte bit – hvad med bytes? I maskinkode (og i BASIC) virker de på hver enkelt bit uafhængig af hinanden. Så for at finde

1 0 0 1 0 1 0 1 EOR 1 1 0 0 1 0 1 1

tager vi først bit 7 (dem til venstre) og regner:

1 EOR 1 = 0

for at regne resultatets bit 7 ud; så gå videre til bit 6:

0 EOR 1 = 1

og så bit 5, 4, 3, 2, 1 og 0:

0 EOR 0 = 0

1 EOR 0 = 1

0 EOR 1 = 1

1 EOR 0 = 1

0 EOR 1 = 1

1 EOR 1 = 0

og når vi læser dem som én linje har vi svaret:

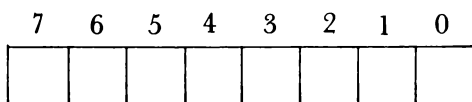
0 1 0 1 1 1 1 0

Tilsvarende med AND og OR.

Opkoderne for de logiske operatoren med alle adresseringsmåder (der er otte) er opført i tillæg 4.

MASKERING

Den måske vigtigste anvendelse af logiske operationer i maskinkodeprogrammering er testning eller ændring af enkelte bit i en byte. Husk at bittene i en 8-bits byte traditionelt nummereres:



sådan at større bit har højere nummer. Hvis jeg nu vil undersøge bit 3 i en byte, hvordan gør jeg så det?

Der er en hel masse bytes som har 1 stående i bit 3; der er nemlig 128, ligesom der er 128 der har 0 i bit 3. Der er ikke noget særligt system i dem hvad angår deres matematiske egenskaber.

Se på denne byte:

0 0 0 0 1 0 0 0

Den har nuller *overalt* undtagen i bit 3, den vi interesserer os for. Lad os kalde dette tal M for *maske* (det svarer jo for øvrigt til decimalværdi 8). Fidusen er nu at ANDe masken M med den givne byte. Bit 7, 6, 5, 4 og 2, 1 og 0 *må* altid være 0 i resultatet, fordi 0 AND et eller andet giver nul. Hvis tallets bit 3 er 0, bliver resultatet

0 0 0 0 0 0 0 0

men hvis bit 3 er 1 bliver resultatet

0 0 0 0 1 0 0 0

Hvis vi sætter M lig med 08 har vi med andre ord:

$p \text{ AND } M = 00$ hvis bit 3 i p er 0

$p \text{ AND } M = 08$ hvis bit 3 i p er 1

Tilsvarende kan vi teste bit 0, 1, . . . , 7 ved at bruge masker:

00000001	01 (hex)	1 (decimal)	for bit 0
00000010	02	2	for bit 1
00000100	04	4	for bit 2
00001000	08	8	for bit 3
00010000	10	16	for bit 4
00100000	20	32	for bit 5
01000000	40	64	for bit 6
10000000	80	128	for bit 7

Måske er vi ikke så interesserede i at *finde* værdien af bit 3, men vi vil gerne sætte den til 0. Differencen kan udtrykkes:

$p - (p \text{ OR } 08)$

som skubber det ciffer ud. Der findes andre smarte variationer af maskeringer, men når man først har fået fat i princippet er det let nok at se hvordan de virker.

En usædvanlig og bemærkelsesværdig mulighed på 64'eren er brugen af sprites – store farvede grafikblokke, som kan flyttes rundt på skærmen og passerer over og under hinandlen. I BASIC er det vanskeligt at få dem til at bevæge sig særlig hurtigt. Med maskinkode er det anderledes: det er hårdt arbejde at få dem til at sætte farten ned!

19 · SPRITES

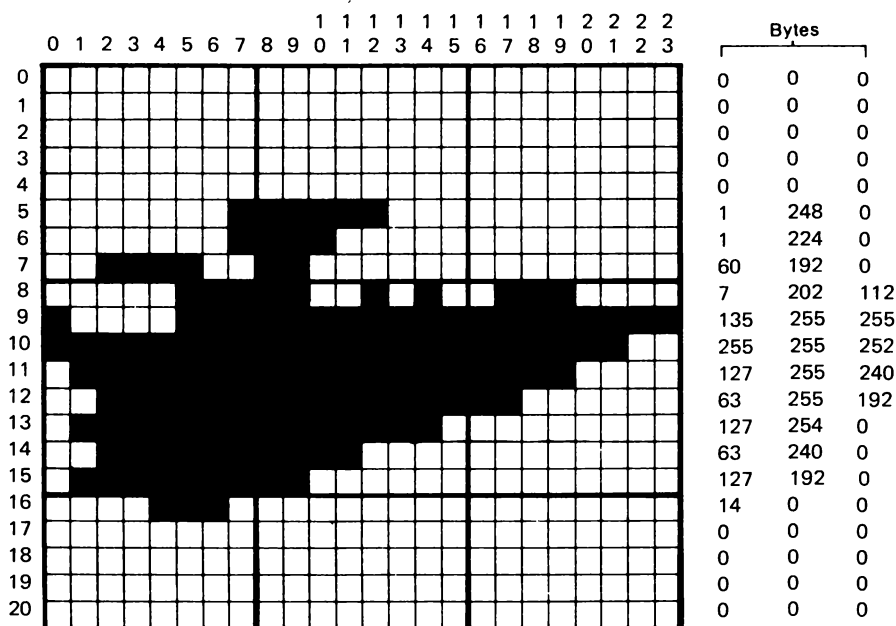
Sprites betyder små åndevæsner. På computeren er det forholdsvis store grafikfigurer, som behandles af en speciel VIC-chip og kan flyttes omkring på skærmen som programøren ønsker. De kan være grundideen i mange spændende spil og skærbilleder. Det er ikke helt lige ud ad landevejen at have med dem at gøre, men i dette kapitel præsenteres du for nogle af grundprincipperne – nok til at du selv kan bruge sprites.

Jeg vil gerne begynde med en almindelig oversigt over de vigtigste teknikker for spritehåndtering, for selv garvede BASIC-programmører kan det forekomme lidt indviklet.

SPRITEKONSTRUKTION

De informationer som definerer en sprite består af et gitter på 21×24 felter, hvor de enkelte felter enten er blanke eller udfyldte. Figur 3 viser fx en sprite i form af en 'stjernekruser'.

De blanke eller udfyldte felter må konverteres til en række tal, som skal lagres på et dertil beregnet sted (se senere). Det gør man ved at erstatte hvert blankt felt med et 0 og hvert udfyldt med 1. Tag hver linje med 24 cifre og split det



Figur 3

op i tre 8-cifrede dele. På den måde bliver for eksempel linje 8 i figur 3 til

00000111 11001010 01110000

Det ser jo ud som almindelige binærtal – og det er netop det der er tanken. Omregnet til decimal giver de

7 202 112

Hver enkelt linje af en sprite kan man altså tænke sig som en serie på tre decimaltal (mellem 0 og 255). Værdierne for hele spriten er anført ned langs højre side af figur 3, og de læses traditionelt i rækkefølge fra øverste venstre til nederste højre hjørne: først tre bytes for linje 0, så tre for linje 1 osv. til række 20. Det giver 63 tal i alt.

Man *kan* tegne sine sprites på ternet papir og regne decimalværdierne ud i hånden, men var det ikke meget bedre at få computeren til at gøre alt det hårde arbejde?

COMPUTERUNDERSTØTTET SPRITEKONSTRUKTION

Her er der et ret enkelt program som gør det muligt at designe en sprite på skærmen og generere datalisten. For at holde programmet inden for visse grænser har jeg sprunget forskellige forbedringsmuligheder over. Hvis du gerne vil gøre programmet mere avanceret, så klø bare på!

```
7010 POKE 53280,4
7020 PRINT CHR$(147)
7030 FOR S=0 TO 20
7040 IF S=8*INT(S/8) THEN PRINT "-----
-----"
7050 IF S<>8*INT(S/8) THEN PRINT ".....:.....
...:.....:"
7060 NEXT S
7080 PRINT:PRINT:PRINT
7100 DIM S(20,23)
7110 FOR L=0 TO 20
7120 FOR K=0 TO 23
7130 CDE=63:GOSUB 8000
7140 GET A$
7150 IF A$<>"0" AND A$<>"1" THEN 7140
7160 IF A$="0" THEN S(L,K)=0:CDE=32:GOSUB 8000
7170 IF A$="1" THEN S(L,K)=1:CDE=102:GOSUB 8000
7180 NEXT K
7190 NEXT L
7200 POKE 53280,3
7210 GET A$:IF A$<>"N" AND A$<>"J" THEN 7210
7220 IF A$="N" THEN POKE 53280,4:GOTO 7110
```

```

7250 PRINT CHR$(19);
7260 FOR L=0 TO 20
7270 FOR X=0 TO 16 STEP 8
7280 V=0
7290 FOR K=0 TO 7
7300 IF S(L,X+K)=1 THEN V=V+2↑(7-K)
7310 NEXT K
7320 PRINT TAB(24+X/2);V;
7330 NEXT X
7340 PRINT
7350 NEXT L
7360 GOTO 7360
8000 POKE 1024+40*L+K,CDE
8010 RETURN

```

RUN det. Kantfarven bliver violet, af en grund som vil vise sig om lidt. Du får et 21×24 gitter af prikker og streger, for nemheds skyld arrangeret i stykker på 8×8 . Der står et spørgsmålstegn øverst til venstre. Hvis du trykker 1 erstattes det af et ternet mønster, hvis du trykker 0 erstattes det af mellemrum. Så rykker det en plads til højre. Du kan fortsætte på denne måde indtil hele gitteret er fyldt.

Så bliver kanten cyanfarvet for at minde dig om at du skal trykke på en tast (der er ikke meget plads til at give meddelelser, og så er det en nem måde at klare problemet på). 'J' for 'ja' får programmet til at fortsætte, 'N' for 'nej' betyder at du har lavet en fejl og gerne vil prøve igen. (Når du gør det om skal du trykke på alle nullerne og ettallerne igen – et punkt hvor det ville være muligt at lave forbedringer).

Computeren udskriver så automatisk data for linjerne ned langs højre side. Skriv dem ned på papir (eller skriv dem ud på en printer, eller kopiér dem til en fil på kassette eller diskette).

Jeg har her noteret tallene i decimal, men du kan naturligvis omregne til hex. Det vigtigste at være klar over er at

det går helt fint at *indlæse* spritedata med BASIC – det er først når det drejer sig om at flytte med sprites at maskinkode bliver uundværlig. For at gøre tingene så enkle som muligt vil jeg bruge BASIC hvor jeg kan.

SPRITEREGISTRENE

Særlige dele af hukommelsen er reserveret for spritehåndtering. Adresserne begynder på 53248 decimal (D000 hex) og slutter på 53294 (D02E). Fra nu af vil jeg bruge hexadresser, da vort endemål jo er maskinkode. Det er ikke alle spriteregistre der er af betydning for begynderen, så jeg vil springe de mere avancerede over. Derudover er der flere *pointere* på adresserne 07F8-07FF, som oplyser computeren om hvor den kan finde de 63 grafikdata der skal til for at danne en sprite. Jeg skal straks beskrive dem nærmere, men først en hurtig gennemgang:

Spritepositioner

Adresse D000-D00F indeholder kolonnennummeret (eller X-koordinaten ved højopløsning) og linjenummeret (Y-koordinaten) for hver af de otte sprites. Numrene går fra 0 til 255. Hvert nummer ligger som en byte i en adresse.

Offsetflag

De otte bit i en enkelt byte på adresse D010 angiver et offset til højre for X-koordinaten (kolonnennummeret). Hvis bit nummer N er sat, lægges der 256 til kolonne nummer N. Det er nødvendigt for at anbringe sprites langs skærmens højre kant.

Enable/disable

De otte bit i en enkelt byte på adresse D015 tænder (enable)

den N'te sprite hvis bit N er 1, og slukker den (disable) hvis N er 0.

Lodret udvidelse

De otte bit i en enkelt byte på adresse D017 udvider den N'te sprite til dobbelt højde hvis bit N er 1.

Vandret udvidelse

Tilsvarende udvider de otte bit på adresse D01D sprite N til dobbelt bredde hvis bit N er 1.

Kollisionsflag

Hvis to sprites 'kolliderer' sættes de tilsvarende bit i D01E til 1.

Farver

Hver af adresserne fra D027 til D02E indeholder farvekode (0-15) for en sprite (se kapitel 16).

Datapointere

Adresserne 07F8-07FF indeholder pointere til startadresserne for data for sprites 0-7. Hvis den N'te pointer har værdien PTR, så begynder dataadresserne på $64 \times \text{PTR}$. Vi vil kalde området fra $64 \times \text{PTR}$ til $64 \times \text{PTR} + 63$ for den PTR'te blok i hukommelsen. Det giver mulighed for at definere sprites overalt inden for de første 16384 bytes RAM. Der er udveje for at bruge de øvrige 49152 bytes, men de er indviklede; se *Programmer's Reference Guide* s. 101 og 133. Men man kan ikke bare lade sprites ligge i gamle adresser; BASIC-systemet vil smadre data. Se de anbefalede adresser nedenfor.

Adresserne for spritekontrol er sammenfattet i tabel 3 og 4 (og i tillæg 7). Om betydningen af de udeladte adresser: se *Programmer's Reference Guide* s. 131-181. Det er 50 sider. Jeg sagde jo at det ikke er lige ud ad landevejen med sprites!

SPRITEKOORDINATER

Linje- og kolonnenumrene for sprites ligger inden for disse decimalværdier:

Linje: 0–255

Kolonne: 0–511 (inklusive offsetflag)

Dette område er meget større end skærmområdet, som er 200 linjer à 320 kolonner. 64'eren drager fordel af det ved at lade dig positionere sprites uden for skærmen og føre dem glidende ind. Sammenhængen mellem spritekoordinater og skærm vises i figur 4.

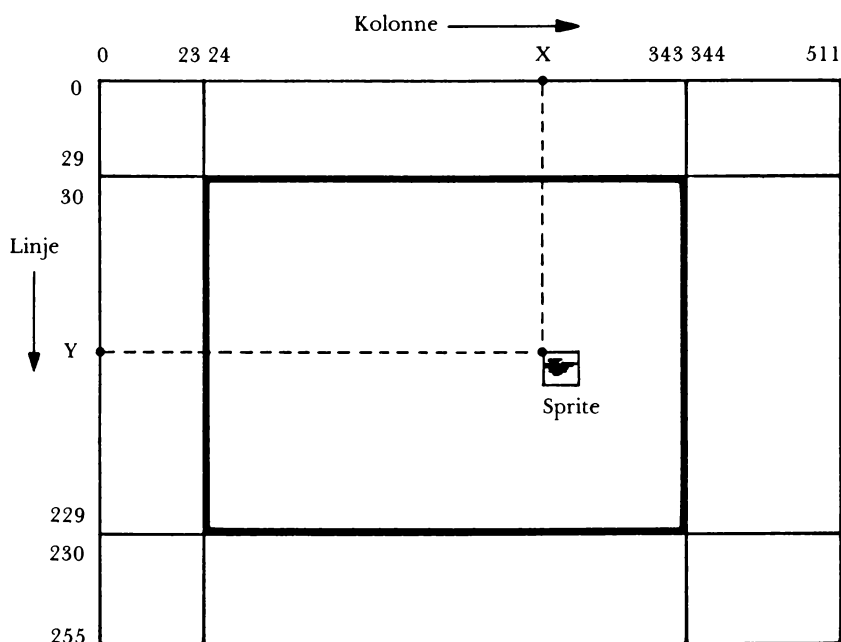
Bemærk at linje- og kolonnenummeret for en sprite er lig med øverste venstre hjørne af *gitteret* som definerer den (de 21×24 felter). Det må man tage med i betragtning i sine programmer. Ved udvidede sprites er det også øverste venstre hjørne.

Tabel 3 Spritedatapointere

Adresse	Indhold
07F8	Datapointer for sprite 0
07F9	Datapointer for sprite 1
07FA	Datapointer for sprite 2
07FB	Datapointer for sprite 3
07FC	Datapointer for sprite 4
07FD	Datapointer for sprite 5
07FE	Datapointer for sprite 6
07FF	Datapointer for sprite 7

Tabel 4 Spriteregistre

Adresse		Indhold								Funktion
D000		Kolonnenummer for sprite 0								Spritepositioner
D001		Linjenummer for sprite 0								
D002		Kolonnenummer for sprite 1								
D003		Linjenummer for sprite 1								
D004		Kolonnenummer for sprite 2								
D005		Linjenummer for sprite 2								
D006		Kolonnenummer for sprite 3								
D007		Linjenummer for sprite 3								
D008		Kolonnenummer for sprite 4								
D009		Linjenummer for sprite 4								
D00A		Kolonnenummer for sprite 5								
D00B		Linjenummer for sprite 5								
D00C		Kolonnenummer for sprite 6								
D00D		Linjenummer for sprite 6								
D00E		Kolonnenummer for sprite 7								
D00F		Linjenummer for sprite7								
D010		Sp 7	Sp 6	Sp 5	Sp 4	Sp 3	Sp 2	Sp 1	Sp 0	Offsetflag
D015		Sp 7	Sp 6	Sp 5	Sp 4	Sp 3	Sp 2	Sp 1	Sp 0	Enable/disable
D017		Sp 7	Sp 6	Sp 5	Sp 4	Sp 3	Sp 2	Sp 1	Sp 0	Lodret udvidelse
D01D		Sp 7	Sp 6	Sp 5	Sp 4	Sp 3	Sp 2	Sp 1	Sp 0	Vandret udvidelse
D01E		Sp 7	Sp 6	Sp 5	Sp 4	Sp 3	Sp 2	Sp 1	Sp 0	Kollisionsflag
D027		Farvekode for sprite 0								Farver
D028		Farvekode for sprite 1								
D029		Farvekode for sprite 2								
D02A		Farvekode for sprite 3								
D02B		Farvekode for sprite 4								
D02C		Farvekode for sprite 5								
D02D		Farvekode for sprite 6								
D02E		Farvekode for sprite 7								



Figur 4

KONSTRUKTION AF EN SPRITE

For at få en sprite på skærmen må vi

1. Definere data for dens form.
2. Sætte pointerne på det sted hvor data er lagret.
3. POKE data ind.
4. Aktivere spriten.
5. Angive spritens farve.
6. Angive linje- og kolonnenumre for spriten.

Lad os tage stjernekruderspriten ovenfor og oprette den som sprite 1. Det kan gøres på denne måde:

```

9000 V=53248
9100 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
9110 DATA 1,248,0,1,224,0,60,192,0,7,202,112,
      135,255,255
9120 DATA 255,255,252,127,255,240,63,255,192,
      127,254,0
9130 DATA 63,240,0,127,192,0,14,0,0,0,0,0,0,
      0,0
9140 DATA 0,0,0,0,0,0
9200 POKE 2041,13      pointer for sprite 1 til blok 13; 2041 = 07F9 hex
9210 FOR G=0 TO 62
9220 READ H            læs data
9230 POKE 832+G,H      POKE til blok; bemærk 832 = 64 × 13
9240 NEXT G
9250 POKE V+21,2       aktivér sprite 1
9260 POKE V+40,7       sprite 1 gul
9270 POKE V+2,100      sprite 1 i kolonne 100
9280 POKE V+3,100      sprite 1 i linje 100

```

Skriv dette *omhyggeligt* ind og RUN 9000; så skulle du gerne se stjernekrudseren i gult som vi har bestilt det.

Du kan eksperimentere med at ændre positionerne med direkte instruktioner:

```
POKE V+2,110
```

flytter den til højre,

```
POKE V+3,90
```

flytter den op,

```
POKE V+40,5
```

gør den grøn. Prøv med andre værdier og se hvad der sker.

BEVÆGELSE

Man kunne selvfølgelig have lavet det samme i maskinkode. Men som sagt: det er udmærket at foretage de indledende manøvrer i BASIC. Men for at få rimeligt hurtig bevægelse må vi bruge maskinkode.

Jeg vil bruge nogenlunde den samme kode, men med små ændringer til hver lejlighed. Ændringerne kommer i begyndelsen, hvilket er uheldigt når man bruger **LOADER**. Derfor vil jeg bruge et lille trick: **No OPeration**-kommandoen:

NOP (opkode EA)

Det betyder 'ingen udførelse', 'se bort fra denne instruktion'. En blok med NOPper vil give os et programområde som vi med EDIT kan lægge flere bit ind i. I øvrigt er NOP helt uskadelig (selv om det forårsager en ubetydelig forsinkelse i starten).

Læg en blok på 12 NOPper (et tilfældigt og mere end rigeligt antal) i starten af programmet:

NOP	EA
.	.
.	.
.	.
NOP	EA

og så kommer der lidt kød på:

LDX #00	A2 00	
loop: STX D002	8E 02 D0	X-koordinat for sprite 1
JSR forsink	20 1B C0	
CLC	18	
INX	E8	

CPX #00	E0 00
BNE loop	D0 F4
RTS	60

Det er ret sandsynligt at vi får brug for en forsinkelsesløkke, derfor JSR forsink. Første gang prøvede jeg det med den lange forsinkelse (X-løkke inden i Y-løkke), men det viste sig at være for optimistisk, og spriten humpede af sted som en enbenet skildpadde. Så jeg vil foreslå dig at bruge den korte version:

forsink:	TXA	8A	
	PHA	48	
	LDX #80	A2 80	passende lang løkke
floop:	DEX	CA	
	CPX #00	E0 00	
	BNE floop	D0 <u>FB</u>	
	PLA	68	
	TAX	AA	
	RTS	60	

Gør nu parat til bevægelsen ved at tilføje BASIC-linjerne

```
9800 GET A$:IF A$="" THEN 9800
9810 SYS(49152)
```

og RUN 9000.

Du ser stjernekyrdsere blive bygget op i gult. Tryk en tast, og den forsvinder hastigt fra midten af skærmen og su-ser af sted fra venstre til højre, og standser så omkring tre-kvartvejs over skærmen. Det er faktisk fordi dens horisonta-le koordinat nu er 255, den højeste værdi vi kan benytte når vi kun bruger adresse D002. Jeg skal om et øjeblik vise dig hvordan vi kan ordne det, men først vil vi se hvordan man laver en sprite større.

UDVIDELSE

For at gøre en sprite dobbelt så bred skal du EDITere de fem første programbytes og ændre dem fra EA til

```
LDA #02      A9 02
STA D01D     8D 1D D0
```

Start nu programmet igen, og spriten vil blive strakt vandret ud. EDITér de næste tre bytes sådan:

```
STA D017     8D 17 D0
```

så bliver den også dobbelt så høj.

Hvad gjorde vi? Vi satte bit 1 i registrene D01D og D017 til 1 ved at lagre $2 \uparrow 1 = 02$. I tabel 4 kan du se at det er de registre der kontrollerer udvidelser.

BEVÆGELSE OVER HELE SKÆRMEN

Hvis du vil placere din sprite på højre side af skærmen, efter kolonne 255, skal du bruge *offsetflaget* i D010. Hvis dettes bit N er sat til 1, lægges der 256 til kolonnennummeret for sprite N. Her er der et eksempel. LOAD det som sædvanlig fra C000 (uden NOPper denne gang!):

```
      LDA D0 10      AD 10 D0 }
      AND FD         29 FD   }  brug maskering med FD
      STA 10 D0      8D 10 D0 }  (11111101 binær)
                                }  for at nulstille offset for sprite 1
      LDY #00        A0 00
      LDX #00        A2 00
loop: STX D002       8E 02 D0
      JSR forsink    20 28 C0
      INX            E8
      CPX #00        E0 00
      BNE loop       D0 F5
```

	INY	C8	
	CLC	18	
	INC D010	EE 10 D0	} læg 2 til offset for at sætte dens bit 2 til 1 og således flytte sprite 1 256 kolonner
	INC D010	EE 10 D0	
	CPY #02	C0 02	
	BNE loop	D0 <u>E9</u>	
	SEC	38	
	DEC D010	CE 10 D0	
	RTS	60	
forsink:	TXA	8A	
	PHA	48	
	LDX #00	A2 00	udskift 00 med 80 for hurtigere bevægelse
floop:	DEX	CA	
	CPX #00	E0 00	
	BNE floop	D0 <u>FB</u>	
	PLA	68	
	TAX	AA	
	RTS	60	

RUN 9000 som sædvanlig. Denne gang suser spriten tværs over skærmen og forsvinder ud over højre kant (et lille stykke af den stikker ud på venstre side til slut; det kan du forhindre ved at nulstille enable/disable-flaget, bit 2, hvis du vil eksperimentere).

TASTATURKONTROL

Vi har allerede set hvordan tastaturet kan aflæses i maskincodeprogrammer, så lad os modificere ovenstående rutine, så den giver os kontrol over stjernekydserens lodrette position på TV-skærmen. Tast 'O' betyder 'op', 'N' betyder 'ned'; og 'S' er 'stop', for jeg vil få krydseren til at flyve hen over skærmen hele tiden. Nu er der kommet et par ekstra finter til koden (som ligger fra C000):

start:	CLC	18	} undersøg om tast 'S' (kode 0D) er nedtrykket, i så fald RTS
	LDA #0D	A9 0D	
	CMP C5	C5 C5	
	BNE spring	D0 <u>01</u>	
	RTS	60	
spring:	LDA D010	AD 10 D0	
	AND FD	29 FD	
	STA D010	8D 10 D0	
	LDY #00	A0 00	
	LDX #00	A2 00	
loop:	STX D002	8E 02 D0	
	JSR forsink	20 <u>35</u> <u>C0</u>	} underrutineadresser udregnet og tilføjet efter at programmet er skrevet
	JSR taster	20 <u>41</u> <u>C0</u>	
	INX	E8	
	CPX #00	E0 00	
	BNE loop	D0 <u>F2</u>	
	INY	C8	
	CLC	18	
	INC D010	EE 10 D0	
	INC D010	EE 10 D0	
	CPY #02	C0 02	
	BNE loop	D0 <u>E6</u>	
	SEC	38	
	DEC D010	CE 10 D0	
	JMP start	4C 00 C0	begynd forfra
forsink:	TXA	8A	underrutine på C035
	PHA	48	
	LDX #00	A2 00	
floop:	DEX	CA	
	CPX #00	E0 00	
	BNE floop	D0 <u>FB</u>	
	PLA	68	
	TAX	AA	
	RTS	60	
taster:	LDA #27	A9 27	underrutine på C041

	CMP C5	C5 C5	undersøg for tast 'N'
	BNE spring1	D0 <u>03</u>	
	INC D003	EE 03 D0	forhøj linjenummeret med 1
spring1:	LDA #26	A9 26	tast 'O'
	CMP C5	C5 C5	
	BNE spring2	D0 <u>03</u>	
	DEC D003	CE 03 D0	
spring2:	RTS	60	

Hvis du nu starter med GOTO 9000 eller RUN 9000 og trykker en tast vil krydseren blive ved at drage hen over skærmen. Tast 'O' for op, 'N' for ned.

Det går ekstremt hurtigt! Når du har prøvet nok vil tast 'S' stoppe legen. Hold 'S' nede et par sekunder.

SPRITEPRIORITET

Hvis to sprites overlapper hinanden, vil den med det laveste nummer se ud som om den ligger oven på den anden. Den 'nederste' vil dog kunne ses gennem 'hullerne' i den øverste, ligesom man ville forvente i det virkelige liv.

For at kunne prøve det vil jeg nu oprette en ny sprite. Vi bruger den samme rutine (det er godt nok i øjeblikket, men senere vil jeg foreslå en bedre fremgangsmåde hvis du skal bruge *mange* sprites):

```

9400 DATA 0,255,0,3,255,192,15,195,240,63,0,
      252,255,0,255
9410 DATA 63,0,252,127,195,254,31,255,248,3,
      255,192
9420 DATA 0,255,0,0,195,0,1,129,128,3,0,192,
      6,0,96
9430 DATA 15,0,240,15,0,240,7,129,224,3,195,
      192

```

```

9440 DATA 1,231,128,0,0,0,31,255,248
9500 POKE 2040,14      pointer for sprite 0 til blok 14
9510 FOR G=0 TO 62
9520 READ H
9530 POKE 896+G,H      blok 14: 896 = 64 × 14
9540 NEXT G

```

For at aktivere både sprite 0 og sprite 1 må vi ændre linje 9250 ovenfor til

```
9250 POKE V+21,3
```

fordi 3 = 00000011 i binær; bit 0 og bit 1 står altså begge på 1. Nu fortsætter vi:

```

9560 POKE V+39,5      sprite 0 grøn
9570 POKE V,120        sprite 0 i kolonne 120
9580 POKE V+1,95      sprite 0 i linje 95
9590 POKE V+29,3      udvid sprite 0 og sprite 1 vandret
9600 POKE V+23,1      udvid sprite 0 lodret

```

(Også her kunne man lave en masse af det i maskinkode, men BASIC er bedre at demonstrere det med. Men måske kunne du have lyst til, som en øvelse, at udarbejde en maskinkoderutine for linje 9560–9600.)

Du skulle stadig have det tidligere stykke maskinkode – det med tastaturkontrollen – i hukommelsen. Med RUN 9000 kan du få stjerne krydseren til at passere det andet monstrum ved skønsom brug af O- og N-tasterne. Kan du se hvordan den går *bagom*? Det er fordi det grønne objekt (sprite 0) har prioritet over krydseren (sprite 1).

Vil vi nu have krydseren til at passere *foran* den grønne tingest må vi ændre prioriteten. En enkel løsning er at lade den grønne tingest skifte prioritet fra 0 til 2. Det medfører følgende ændringer:

9500 POKE 2042,14

9250 POKE V+21,6 6 = 00000110

9560 POKE V+41,5

9570 POKE V+4,120

9580 POKE V+5,95

9590 POKE V+29,6

9600 POKE V+23,4

Prøv det igen – krydseren går foran, ikke bagved.

FLERE SPRITES MED ENS DATA

Vi kan oprette flere sprites med de samme data ved at give to eller flere pointere samme værdi. Hvis vi har sprite 1 og sprite 2 som ovenfor, men nu vil have sprite 0 til at være en sort tingest (også med dobbelt størrelse) på en anden position, så aktiverer vi alle tre sprites ved at ændre 9250 en gang til:

9250 POKE V+21,7 7 = 00000111

Opret nu sprite 0

9700 POKE 2040,14	data for sprite 0 fra samme blok: 14
9760 POKE V+39,0	sprite 0 sort
9770 POKE V,70	sprite 0 i kolonne 70
9780 POKE V+1,124	sprite 0 i linje 124
9790 POKE V+29,7	alle 3 sprites udvidet vandret
9795 POKE V+23,5	kun 0 og 2 udvidet lodret

Hvis du RUNner nu ser du to tingester plus en krydser.

REGISTRERING AF KOLLISIONER

Med kollisionsregistret, D01E hex (eller V + 30 hvor V = 53248 som i vore BASIC-programmer) kan man afgøre når der sker kollision mellem sprites. Hvis to sprites kolliderer bliver de tilsvarende bit sat til 1. Hvis det fx er sprite 1 og sprite 2 der kolliderer vil D01E indeholde

00000110 = 06

Denne værdi opdateres ved hver kollision. Hvis du skal teste for kollision mellem sprite 1 og sprite 2 skal du altså bruge en stump maskinkode som denne:

```
LDA #06
CMP 1E D0
BEQ virkning
```

·
·
·

virkning: hvad du nu vil have der sker når de kolliderer

Adresse D01F (V + 31) reagerer på en kollision mellem sprites og tekst i forgrundsfarve: bit N er sat hvis sprite N kolliderer (i *User's Reference Guide* står der 'Sprite to Background Collision'; der skulle have stået 'Sprite to Foreground Collision').

HVOR SPRITEDATA SKAL LAGRES

Til tre sprites eller færre kan man bruge blok 13, 14 og 15. De ligger i virkeligheden i *kassettebufferen*, et område af hukommelsen som kun benyttes når kassettebåndoptageren arbejder. Så det er et sikkert sted at gemme sine sprites. Desværre er den ikke lang nok til at indeholde alle otte

64-bytes blokke. Så må man prøve andre steder. *User's Reference Guide* foreslår blok 192-199, hvis du ikke har et meget langt BASIC-program. Hvis du vil vide mere kan du som sædvanlig konsultere *User's Reference Guide*.

DET VAR BARE BEGYNDELSEN

Det var et langt kapitel, og vi har bare lige kradset i overfladen: Man kan for eksempel have flerfarvede sprites. Men ét sted må jeg standse, og jeg håber du har fået ideer nok til at holde dig beskæftiget. Når du først mestrer hvad jeg har fortalt om spritehåndtering kan du altid kigge videre i *User's Reference Guide* efter andre muligheder, som ligger uden for denne bogs rammer.

Din håndbog fortæller hvordan du bruger grafikkarakterer, men den nævner ikke at 64'eren kan præsentere noget så spændende som:

20 · HØJOPLØSNINGSGRAFIK

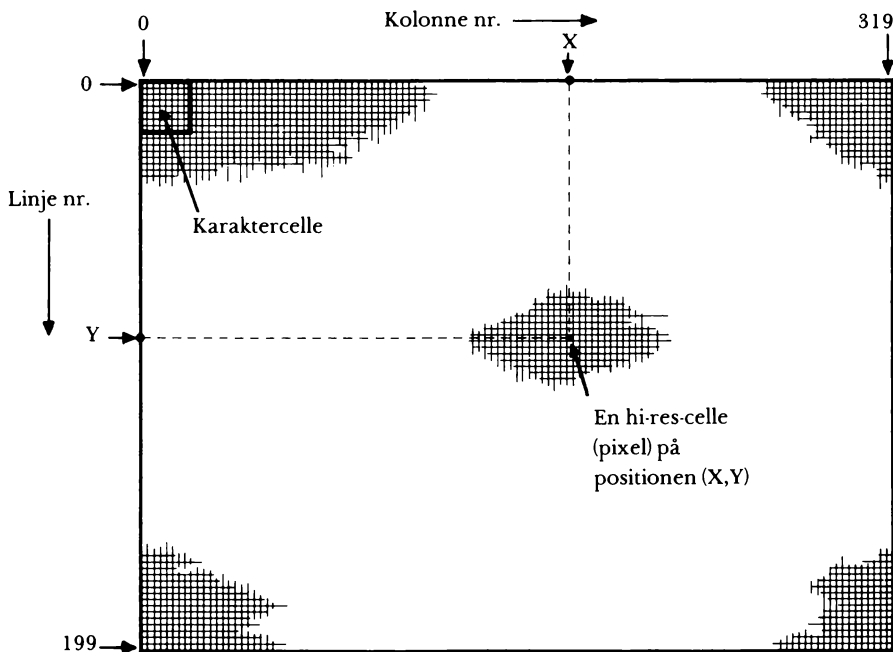
Hver enkelt karaktercelle på TV-skærmen er i virkeligheden dannet af et kvadrat på 8×8 mindre celler eller *pixels*, de punkter som karakteren (et sted langt inde i elektronikken) bliver opbygget af. Ved at skaffe sig direkte adgang til disse celler kan man plote grafiske billeder på en *hi-res*-skærm (højopløsningsskærm, modsat *lo-res*: lavopløsning). Det vil sige at man har et billede på $25 \times 8 = 200$ linjer og $40 \times 8 = 320$ kolonner. Det er næsten det samme nummersystem som bruges ved sprites, men er begrænset til skærmområdet (se figur 5).

HI-RES MODE

Før du kan få din maskine til at lave højopløsningsgrafik må du sætte den i *hi-res* mode (højopløsningsindstilling), oprette et område til de grafiske data i hukommelsen og rense ud i det område. Farver skal også angives. *Programmer's Reference Guide* forklarer det på side 123. Her er der et BASIC-program (så du kan se hvad der sker) som giver en ren skærm i lysegrønt. Hvis du retter 13-tallet i 11080 til

16 * SKRIFT + PAPIR

hvor SKRIFT og PAPIR er koderne for forgrunds- og baggrundsfarve, kan du få en hvilken som helst kombination af



Figur 5

farver. Den følgende rutine vil lægge skærmhukommelsen i området begyndende med adresse 8192:

```

11000 REM HI-RES INITIALISERING
11010 POKE 53265, PEEK(53265) OR 32
11020 POKE 53272, PEEK(53272) OR 8
11030 BM=8192
11040 FOR U=BM TO BM+7999
11050 POKE U,0
11060 NEXT U
11070 FOR U=1024 TO 2023
11080 POKE U,13
11090 NEXT U
11100 RETURN

```

Start programmet med GOSUB 11000. Først får du noget v s, s  en ren, mest sort baggrund, men med nogle farvede klatter hvor teksten stod, s  renses sk rmen helt og bliver lysegr n (pr v at erstatte de 13 i linje 11080 med 16 * SKRIFT + PAPIR hvor SKRIFT og PAPIR er koderne for de farver du vil have; programmet giver sort skrift p  lysegr nt papir).

Du vil bem rke at sk rmrensningen tager temmelig lang tid: omkring 20 sekunder i BASIC.

OG NU I MASKINKODE

Da BASIC er s  langsomt har du her det samme program omskrevet til maskinkode:

```

        LDA D011      AD 11 D0
        ORA #20        09 20
        STA D011      8D 11 D0
        LDA D018      AD 18 D0
        ORA #08        09 08
        STA D018      8D 18 D0
        LDA #00        A9 00
        STA FB         85 FB
        LDA #20        A9 20
        STA FC         85 FC
        LDY #00        A0 00
loop:   LDA #00        A9 00
        STA (FB),Y     91 FB
        INC FB         E6 FB
        CMP FB         C5 FB
        BNE spring    D0 02
        INC FC         E6 FC
spring: LDA #3F        A9 3F
        CMP FB         C5 FB
        BNE loop      D0 EE

```

	CMP FC	C5 FC
	BNE loop	D0 <u>EA</u>
	LDA #00	A9 00
	STA FB	85 FB
	LDA #04	A9 04
	STA FC	85 FC
	LDY #00	A0 00
loop2:	LDA #0D	A9 0D
	STA (FB),Y	91 FB
	INC FB	E6 FB
	LDA #00	A9 00
	CMP FB	C5 FB
	BNE spring2	D0 <u>02</u>
	INC FC	E6 FC
spring2:	LDA #E7	A9 E7
	CMP FB	C5 FB
	BNE loop2	D0 <u>EC</u>
	LDA #07	A9 07
	CMP FC	C5 FC
	BNE loop2	D0 <u>E6</u>
	RTS	60

Hvis du starter dette med SYS(49152) vil du se at skærmen renses i en håndevending!

PLOT

Kolonner og linjer i hi-res definerer et koordinatsystem på TV-skærmen som vist i figur 5. Hovedopgaven er at finde en måde til at *plotte* et enkelt punkt i kolonne X linje Y – altså koordinaten (X,Y). Ved at kombinere sådanne plotninger kan vi tegne linjer og kurver og udfylde hele store områder. Følgende BASIC-rutine kan tegne et enkelt punkt i linje Y kolonne X. Det går ud fra at Y er mellem 0 og 199, og at X

er mellem 0 og 320. Hvis du vil vide hvordan det virker, så se *Programmer's Reference Guide*.

```
12000 REM PLOT X,Y
12010 BY=8192+320*INT(Y/8)+8*INT(X/8)+(Y AND 7)
12020 BT=7-(X AND 7)
12030 POKE BY,PEEK(BY) OR (2↑BT)
12040 RETURN
```

Idet jeg går ud fra at du stadig har maskinkoden 'slet hi-res-skærm' på plads i C000, har du her et eksempel på hvordan hi-res-plotning bruges:

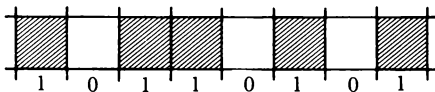
```
13000 SYS(49152)
13010 FOR X=0 TO 319
13020 Y=100+80*SIN(X/10)
13030 GOSUB 12000
13040 NEXT
```

Sammen med underrutinen *plot* giver det en bølgeformet sinuskurve. Andre kurver fås ved at ændre linje 13020.

HVORDAN VIRKER DET?

Dette afsnit bliver lidt teknisk, så hvis du vil kan du springe det over og vende tilbage til det senere.

Hver byte i hi-res-skærmhukommelsen indeholder data for en linje med 8×1 punkter på hi-res-skærmen. Et binært 0 betyder 'ingen prik her'. Byten 10110101 for eksempel giver altså den virkning som er vist i figur 6.



Figur 6

Når du sætter systemvariablen i adresse 53265 til hi-res-mode, instrueres computeren af operativsystemet om at fortolke data på den måde. Det kaldes *bit-mapped* grafik.

Adresserne for vores hi-res skærmhukommelse svarer til de faktiske skærmpositioner som vist i tabel 5

Tabel 5

	0	1	2	...	39	← Kolonnenummer i lo-res
	0 8192	8200	8208	...	8504	0
	1 8193	8201	8209	...	8505	
	2 8194	8202	8210	...	8506	
	3 8195	8203	8211	...	8507	
	4 8196	8204	8212	...	8508	
	5 8197	8205	8213	...	8509	
Linjenummer i hi-res	6 8198	8206	8214	...	8510	
	7 8199	8207	8215	...	8511	
↓	8 8512	8520		...		1
	9 8513	8521		...		
	10 8514	8522		...		
	11 8515	8523		...		
	12 8516	8524		...		
	13 8517	8525		...		
	14 8518	8526		...		
	15 8519	8527		...		
				...		

Disse adresser begynder i 2000 hex. Hver karaktercelle, som før svarede til én adresse i skærmhukommelsen, svarer altså nu til *otte* adresser: en otte bytes lang blok i hukommelsen. Blokkene er ordnede på samme måde som cellerne i

skærmhukommelsen: følg først en lo-res-linje henad, spring så en linje efter kolonne 39.

Nu vil vi tegne en skrå streg i øverste venstre hjørne, fem punkter lang. Adresserne og deres indhold er som i figur 7.

Adresse		Indhold	Decimal	Hex
8192	(2000 hex)	1 0 0 0 0 0 0 0 . .	128	80
8193	(2001)	0 1 0 0 0 0 0 0 . .	64	40
8194	(2002)	0 0 1 0 0 0 0 0 . .	32	20
8195	(2003)	0 0 0 1 0 0 0 0 . .	16	10
8196	(2004)	0 0 0 0 1 0 0 0 . .	8	08
. . . .				

Figur 7

Dette program skulle udføre jobbet:

```

10 GOSUB 11000
20 POKE 8192,128
30 POKE 8193,64
40 POKE 8194,32
50 POKE 8195,16
60 POKE 8196,8
70 GOTO 70

```

Prøv det og se ad.

Samme fremgangsmåde kan anvendes generelt:

1. Find den pågældende adresse.
2. POKE den med værdien for det ønskede billede. Eller brug en STA-maskinkodekommando, som vi senere skal se.

Da vi ikke ønsker at fjerne noget der allerede står på skærmen, må vi gå ud fra at adressen kan indeholde en værdi forskellig fra nul. Det kræver at vi udfører et OR mellem den gamle værdi og den nye (se kapitel 18).

Linje 12010 udregner den korrekte adresse.

Linje 12020 udregner værdien der skal POKEs ind for at plotte ét nyt punkt.

Linje 12030 ORer denne værdi med det eksisterende indhold og POKer resultatet ind i stedet.

For flere detaljer: konsultér *Programmer's Reference Guide* s. 125.

EN PLOTTERUTINE I MASKINKODE

Hvis du ikke er meget ambitiøs vil du sikkert bruge et BASIC-program til at 'drive' hi-res-plotningen med. Men der er ikke nogen grund til at bruge BASIC til selve dens kerne: PLOT X,Y-rutinen. Den vil vi skrive i maskinkode.

Jeg skriver den som en blot og bar rutine, bagefter vil jeg give forslag til hvordan den og rutinen 'rens hi-res-skærm' kan lægges i samme pakke.

Der skal bruges fire databytes:

C000	XL-koord	lille byte af kolonnennummer	(op til
C001	XS-koord	store byte af kolonnennummer	319 i alt)
C002	Y-koord	linjenummer (op til 199)	
C003	test	til brug for fejlfinding	

Programmet begynder altså i C004.

En 16-bit-sammenlægger er ved at blive uundværlig. Først skriver vi en der adderer indholdet i 00FB-00FC med 00FD-00FE og gemmer resultatet i 00FB-00FC. Med zeropage bliver koden enklere:

```

addition: CLC          18
           LDA FB      A5 FB
           ADC FD      65 FD
           STA FB      85 FB
           LDA FC      A5 FC
           ADC FE      65 FE
           STA FC      85 FC
           RTS         60

```

Den virker ganske ligesom 16-bit-sammenlæggeren i kapitel 8, her bare anvendt på side nul. Bemærk at jeg har skrevet den som underrutine (på C004).

Så starter vi på hovedprogrammet, som ligger på C012 (49170 decimal). Vi skal have lavet et modstykke til BASICs

$$BY = BM + 320 * INT(Y/8) + 8 * INT(X/8) + (Y \text{ AND } 7)$$

hvor $BM = 8192 = 2000$ hex. Vi begynder med at få de 2000 i hus:

```

hoved: LDA #20        A9 20
       STA FC         85 FC
       LDA #00        A9 00
       STA FB         85 FB
       STA FD         85 FD

```

Næste opgave er at opbygge $INT(Y/8)$. Det gøres ved højreforskydning tre gange på stribe:

```

LDA Y-koord  AD 02 C0
LSR          4A
LSR          4A
LSR          4A

```

} ikke værd at lave en løkke for!

Så til det mere tricky: at gange med 320; *det* var en værre en! Men der er en lettere måde end direkte multiplikation. Bemærk at $256 + 64 = 320$. Det er nemt nok at gange med

256: flyt lille byte til store byte! Da jeg har været så snedig at lægge #00 ind i FD allerede, er alt hvad vi behøver.

```
STA FE      85 FE
```

Det skal nu lægges til mellemtotalen i FB-FC:

```
JSR addition 20 04 C0
```

Du ville måske tro at $64 \times \text{INT}(Y/8)$ findes ved at fordoble $\text{INT}(Y/8)$ seks gange; men efter det vi allerede har gjort er det lettere at halvere $256 \times \text{INT}(Y/8)$ to gange!

```
CLC          18
LSR FE       46 FE
ROR FD       66 FD
LSR FE       46 FE
ROR FD       66 FD
JSR addition 20 04 C0
```

} hold rede på menterne

Nu har vi hvad der svarer til $\text{BM} + 320 \times \text{INT}(Y/8)$. Så til $8 \times \text{INT}(X/8)$. Det er bare at nulstille bit 0-2 i X, det kan vi klare med maskering. Og vi behøver endda kun arbejde med lille byte af X!

```
LDA XS-koord AD 01 C0
STA FE       85 FE
LDA XL-koord AD 00 C0
AND #F8      29 F8      F8 = 11111000 binær; maskering
STA FD       85 FD
JSR addition 20 04 C0
```

Så er der kun $(Y \text{ AND } 7)$ tilbage i denne del af udregningen, og det kræver ikke den helt store indsats:

```
LDA #00      A9 00
STA FE       85 FE
LDA Y-koord  AD 02 C0
```

```

AND #07      29 07
STA FD       85 FD
JSR addition 20 04 C0

```

Vi har nu afsluttet den del af beregningen, og adressen for lagring af den aktuelle skærm-byte ligger i 00FB-00FC. Sædelt anbragt parat til at bruge postindekseret indirekte adressering (det er nemlig helt gennemtænkt, vil jeg bare sige dig)!

Men (gisp, støn) vi er ikke færdige endnu. Der er den næste del, den dér med BT. Først må vi udregne $7 - (X \text{ AND } 7)$. Også her kræves kun lille byte:

```

LDA XL-koord AD 00 C0
AND #07      29 07
STA FD       85 FD
LDA #07      A9 07
SEC          38
SBC FD       E5 FD
TAX          AA

```

BT står i X-registret.

Jeg har puttet det i X-registret fordi jeg vil bruge det til at kontrollere en løkke som skal opbygge $2 \uparrow$ BT:

```

INX          E8
CLC          18
LDA #01      A9 01
loop: DEX     CA
CPX #00      E0 00
BEQ spring   F0 04
ASL          0A
CLC          18
BCC loop     90 F7

```

} medfører forgrening med offset
(relokérbar kode, ikke JMP)

Nu er alt hvad vi har tilbage at ORe dette med indholdet i skærmhukommelses-byten (indirekte postindekseret adressering gør underværker her) og lagre det igen:

```
spring: LDY #00      AO 00
          ORA (FB),Y   11 FB
          STA (FB),Y   91 FB
```

Under udarbejdelsen tilføjede jeg en linje:

```
test: STA C003      8D 03 C0
```

som gav mig mulighed for med PEEK (49155) at finde ud af hvilken byte der stod som resultat i akkumulatoren. (Og det var da et held at jeg gjorde det, for på første forsøg lavede jeg en gevaldig bommert ved at springe en programlinje over). Du kan godt udelade den, men udelad *aldrig* det afsluttende

```
RTS      60
```

for at komme tilbage til BASIC.

Før du starter denne rutine må du indlæse XL, XS og Y på de rigtige steder i dataområdet (C000, C001, C002) og så bruge

```
SYS(49170)
```

for at starte fra *hovedprogrammet* og ikke fra *additionsprogrammet*.

EN HI-RES-PAKKE

Alt det ovenstående blev fremstillet noget brudstykkevis. Den sidste opgave bliver at sætte stykkerne sammen til en velordnet pakke, som er til at stole på.

For tiden har du plotterrutinen liggende i hukommelsen.

Efter det afsluttende RTS kan du tilføje den skærmsletterutine vi havde før. Det bliver på adresse C074 (eller C071, hvis du udelod testlinjen, som du er i din gode ret til); du kan checke det med LOADERS PRINT-funktion. Nu har du en plotterutine på adresse 49170 og en skærmsletterutine på 49268 (måske). Nu skal du skrive et BASIC-‘drivprogram’, som for eksempel tegner en cirkel:

```
15000 SYS(49268):REM SLET HI-RES-SKAERM
15010 FOR D=0 TO 359
15020 DR= $\pi$ *D/180:REM OMREGNING TIL RADIANS
15030 X=INT(160+90*COS(DR))
15040 Y=INT(100+90*SIN(DR))
15050 XS=INT(X/256):XL=X-256*XS
15060 POKE 49152,XL:POKE 49153,XS:POKE 49154,Y
15070 SYS(49170):REM PLOT X,Y
15080 NEXT
15090 GOTO 15090
```

Du kan selvfølgelig ændre i det på en masse måder. Og du kan skrive maskinkoderutiner på endnu højere adresser til at drive rutinerne *slet skærm* og *plot* med.

LOADER savner en god editor. Men 64'eren har allerede en glimrende editor, nemlig den den bruger til BASIC. Her viser vi dig hvordan man kan narre computeren til at bruge BASIC-editoren til at editere maskinkode i stedet for!

21 · MINIASS – EN HJÆLP TIL GØR- DET-SELV-ASSEMBLERING

Vi må nok indrømme, at den teknik vi hidtil har brugt til at assemblere maskinkode og indlæse den i hukommelsen, er temmelig primitiv.

Du kan selvfølgelig købe en assembler til at gøre hele arbejdet (se kapitel 22), men der er to ulemper ved det. Først og fremmest koster det penge, og for det andet er der fare for at du ikke lærer så meget om hvordan maskinkode virkelig arbejder, fordi assembleren skjuler ting for dig. I alt fald er assemblere på kassette i det store og hele mindre formålstjenlige; man skal faktisk bruge disketteversioner for at få effektive hjælpemidler.

Dette kapitel præsenterer dig for et kompromis: et forbløffende enkelt sæt BASIC-rutiner, som overflødiggør noget af det hårde arbejde, og som gør det betydelig lettere at lokalisere fejl.

EDITOREN

Du vil allerede have bemærket at vi har behov for en enkel måde at editere kode på, at tilføje rutiner, ændre i programmet af hensyn til fejlfinding eller bare at indskrive en glemt instruktion. Men sådan en har vi allerede: BASIC-editoren.

Hvis vi bare på en eller anden måde kunne spænde den for vognen og få den til at editere maskinkode ville halvdelen af vore problemer være løst fra begyndelsen. Her viser det sig så at en egenskab ved Commodore-BASIC, som almindeligvis er en kilde til irritation, pludselig kommer til sin ret. Hvis man skriver et linjenummer med et eller andet volapyk efter, vil BASIC gladeligt sluge det og først brokke sig når det skal udføres. Hvis dette 'volapyk' er maskinkode, som vi aldrig vil prøve at udføre, og vi derimod udfører en startroutine der begynder med et højere linjenummer, så vil alt være i den skønneste orden. Vores kode kunne for eksempel se sådan ud:

```
10 :A2 00      LDX #00
20 :A0 FF      LDY #FF
30 :BD 00 C0    LDA C000,X
40 :09 F0      ORA #F0
50 *
```

Læg mærke til tre ting:

1. Hver linje begynder med et kolon. Det adskiller linjenummeret fra koden, hvilket ikke gør nogen forskel i linje 10-30, men linje 40 uden kolon ville blive opfattet som linje 4009 og altså komme efter linje 50.
2. Alle maskinkode-bytes er adskilt med ét mellemrum – hverken mere eller mindre. Hvis der er to mellemrum eller flere vil programmet tro at instruktionen er slut og ignorere alt derefter. Det giver dig mulighed for at skrive kommentarer på hver linje, fx ved at skrive det tilsvarende i assembler, sådan som jeg har vist det.
3. En asterisk (stjerne) på kolonets plads er en kodeafgrænser: den viser hvor koden slutter.

HVORDAN EN BASIC-LINJE LAGRES

For at få dette til at virke må vi først vide hvordan BASIC-kode oplagres i 64'eren. Det er ikke særlig kompliceret. Det begynder ved 2048 (decimal), som altid indeholder et nul. De to næste bytes indeholder linjenummeret, og så kommer linjens programtekst, afgrænset af en nul-byte.

Her er et eksempel:

Maskin-adresse	Indhold	Betydning
2048	0	altid nul
2049	12	} pointer for næste linje $= 8 \times 256 + 12 = 2060$ linjenummer $= 0 \times 256 + 10 = 10$
2050	8	
2051	10	
2052	0	
2053	58	:
2054	65	A
2055	50	2
2056	32	mellemrum
2057	48	0
2058	48	0
2059	0	linje slut
2060	23	} pointer for næste linje $= 8 \times 256 + 23 = 2071$ linjenummer = 20
2061	8	
2062	20	
2063	0	
2064	58	:
2065	65	A
2066	48	0
2067	32	mellemrum
2068	70	F
2069	70	F
2070	0	linje slut

Skemaet viser hvordan disse to BASIC-linjer lagres:

```
10 :A2 00
20 :A0 FF
```

KODEN

Vi vil lade hovedrutinen begynde på 10000. Alt hvad den skal er at spørge hvor den assemblerede kode skal indlæses, initialisere startadressen (LS) og derpå kalde en rutine der kan behandle en enkelt instruktion (dvs. en linje). Så ændrer den simpelt hen startadressen i henhold til de to linjepointer-bytes og gentager processen.

Instruktionsdekoderen svarer med et flag ved navn SLUT, som er nul (falsk) hvis der ikke er flere instruktioner at behandle, og med -1 (sand) hvis den er stødt på den afsluttede asterisk.

```
10000 INPUT "STARTADRESSE FOR KODE";SA
10010 LS=2049:PB=SA
10020 GOSUB 12000:REM DEKODNING AF EN
      INSTRUKTION
10030 IF SLUT THEN END
10040 LS=PEEK(LS)+256*PEEK(LS+1)
10050 GOTO 10020
```

Instruktionsdekoderen ser sådan ud:

```
12000 REM DEKODNING AF EN INSTRUKTION
12010 PT=LS+4:SLUT=0
12020 IF PEEK(PT)=172 THEN SLUT=-1:RETURN
12030 IF PEEK(PT)=58 THEN GOSUB 14000:RETURN:
      REM INGEN LABEL
12040 GOSUB 16000:RETURN:REM LABEL
```

PT bliver sat til LS + 4 for at hoppe forbi pointeren til næste linje og linjenummeret. PT skulle nu enten pege på en asterisk (172), i så fald er programmet slut, eller på et kolon (58), i så fald skal vi kalde en underrutine på 14000, som håndterer instruktionen hvis der ingen label er. Hvad er det for noget, det der med labels? Labels betyder etiketter, og dem har jeg aldrig fortalt om. Man vil jo gerne holde på etiketten.

Vi vil udskyde dette emne til senere. I øjeblikket vil vi antage at den karakter der følger efter linjenummeret med garanti er enten et kolon eller en asterisk, og vi kan altså ikke nå linje 12040.

Da PT peger på et kolon, må vi forhøje den med 1, så den peger på det første hexciffer af en byte. Så kalder vi en underrutine på 20000, som dekode byten og lagrer den i adresse PB. PB puffes én opad, så den er parat til næste byte, og PT puffes to op, så den peger enten på et mellemrum mellem to bytes eller flere mellemrum på stribe eller på et linjeslut-tal. I de sidste to tilfælde er vi færdige med linjen og kan RETURNere.

```
14000 PT=PT+1
14010 GOSUB 20000:REM DEKODET BYTE TIL PB
14020 PB=PB+1
14030 PT=PT+2
14040 IF PEEK(PT)=0 OR (PEEK(PT)=32 AND
      PEEK(PT+1)=32) THEN RETURN
14050 PT=PT+1
14060 GOTO 14010
```

DEKODNING AF EN BYTE

```
20000 FOR N=0 TO 1
20010 D(N)=PEEK(PT+N)
20020 IF D(N)>64 THEN D(N)=D(N)-7
20030 D(N)=D(N)-48
20040 NEXT N
20050 POKE PB,(D(0)*16+D(1))
20060 RETURN
```

Det skriver næsten sig selv. Vi samler de to alfakoder op på PT og PT + 1. De kan være mellem '0' og '9' (kode 48 til 57) eller mellem 'A' og 'F' (kode 65-70). Vi går nu frem næsten som i kapitel 2. For nemheds skyld vil vi have 'A' til at fortsætte direkte efter '9', fordi det har værdien 10; det opnår vi ved simpelt hen at trække 7 fra alle bogstaver. Nu er alle koder netop 48 større end deres sande værdi, så derfor trækker vi 48 fra. Til slut ganger vi den første værdi med 16 og lægger den næste til for at få decimaltallet svarende til hex-koden og POKEr resultatet ind i PB.

LABELS

Nu virker det hele som en drøm, og man kan tilføje, slette og ændre linjer så tit man lyster, starte LOADER igen, og alt er skønt. Det vil sige: næsten alt. Det eneste problem er hvis der optræder forgreninger rundt omkring i koden; så skal offsetværdierne ændres. Ville det ikke være rart hvis programmet gjorde det for os?

Det medfører at alle forgreningsinstruktioner må mærkes på en eller anden måde, og at den adresse som der forgrenes til må mærkes med den samme label (labels ved mnemonics er omtalt i kapitel 11). For at gøre koden enkel vil jeg sætte nogle skrappe grænser for hvordan en tilladt label kan se ud:

1. Den skal begynde med 'L' (jeg vil senere slække på denne begrænsning).
2. Den skal indeholde præcis to karakterer.

Det er let at se hvorfor jeg laver disse restriktioner. En to-karakters kode ser nogenlunde ud som enhver anden byte, sådan at vi ikke behøver fumle rundt med pointerne, men når den begynder med 'L' kan det ikke være et hexstal, og den kan derfor let udskilles som label.

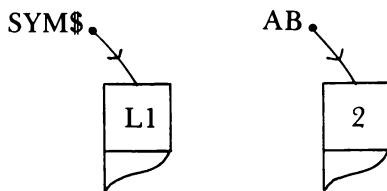
Nu ser et stykke kode for eksempel sådan ud:

```

10      :A2 00
20L1    :A0 FF      Forgrening tilbage hertil
30      :BD 00 C0
40      :D0 L1      BNE L1

```

For at få dette til at virke bliver vi nødt til at foretage nogle modifikationer og tilføjelser. For det første ved vi nu hvorfor der skal være en rutine ved 16000, som tager sig af de tilfælde hvor den pågældende linje hverken har kolon eller asterisk som sin første karakter. For det andet må byte-dekoderen (20000) revideres med henblik på en label i adressefeltet. Endelig må denne rutine vide hvad den skal skrive dér i stedet, og det medfører at vi får brug for en ekstra rutine til, før noget andet sker, at gennemsøge koden for instruktioner der indeholder labels, lægge mærke til hvor de står og etablere et par tabeller der kan holde regnskab med dem, sådan her:



Hvis 'A2' i eksemplet ovenfor betragtes som liggende i byte 0, refererer L1 til byte 2. Symboltabellen (SYM\$) indeholder L1, og det tilsvarende element i AB (antal bytes) er 2.

Med denne fremgangsmåde er det ret ligetil at modificere byte-dekoderen. Linjerne 20050-20060 bliver:

```
20050  KODE=D(0)*16+D(1)
20060  IF KODE<=255 THEN POKE PB,KODE:RETURN
```

På den måde bliver koden der skal POKes beregnet ligesom før, men resultatet kan nu være større end 255 (FF), nemlig hvis første karakter er 'L'. Hvis vi når 20070 har vi altså fundet en label:

```
20070  S$=CHR$(PEEK(PT))+CHR$(PEEK(PT+1))
20080  FOR L=0 TO 50
20090  IF SYM$(L)=S$ THEN 20120
20100  NEXT L
20110  PRINT "LABEL ";S$;" IKKE FUNDET":END
20120  KODE=AB(L)-PB+SA-1
20130  IF KODE>=0 THEN POKE PB,KODE
20140  IF KODE<0 THEN POKE PB,256+KODE
20150  RETURN
```

Linje 20070 fremstiller en label som en streng, som der så søges efter i SYM\$. Når den er fundet peger L på antallet af bytes, dvs. på koden i AB. Linje 20120 udregner så offset. Lad os tage et eksempel, og antage at koden er indlæst fra 50000 og frem. Vi har

```
SA = 50000
PB = 50008 (fordi den peger på L1 i linje 40)
AB(1) = 2
```

Således er $KODE = 2 - 50008 + 50000 - 1 = -7$ lig med antal bytes der skal springes. Men da det er negativt kan det ikke

POKEs direkte. Vi må udregne dets komplement ved at lægge 256 til (linje 20140).

DEKODNING MED LABELROUTINE (16000)

Her har vi et englebarn. Alt hvad vi skal gøre er at flytte pointeren PT frem til kolonet og kalde rutinen 'dekodning uden label':

```
16000 PT=PT+2
16010 GOSUB 14000
16020 RETURN
```

SYMBOLTABELLEN

Alt dette efterlader kun et problem: at opbygge symboltabellen (SYM\$ og AB) til at begynde med.

Før vi kan gøre det må vi tælle alle bytes i programmet, hvilket ikke skulle være så vanskeligt. Antallet af bytes pr. instruktion er én mere end antallet af enkeltstående mellemrum.

I grundtrækkene kommer koden til meget at ligne den række dekodningsrutiner vi allerede har, bortset fra at der ikke sker nogen dekodning:

```
22000 DIM SYM$(50),AB(50)
22010 BC=0:P=0:LS=2049
22020 GOSUB 24000:REM TÆL EN LINJE
22030 IF SLUT THEN RETURN
22040 LS=PEEK(LS)+256*PEEK(LS+1)
22050 GOTO 22020
```


OPTÆLLING AF EN LINJE

```
24000 PT=LS+4:SLUT=0
24010 IF PEEK(PT)=172 THEN SLUT=-1:RETURN
24020 IF PEEK(PT)=58 THEN GOSUB 26000:RETURN:
      REM INGEN LABEL
24030 GOSUB 28000:RETURN:REM LABEL
```

I TILFÆLDE AF 'INGEN LABEL'

I dette tilfælde skal vi bare tælle bytes og forhøje BC tilsvarende:

```
26000 PT=PT+3
26010 IF PEEK(PT)=0 THEN BC=BC+1:RETURN
26020 IF PEEK(PT+1)=32 THEN BC=BC+1:RETURN
26030 BC=BC+1
26040 GOTO 26000
```

I TILFÆLDE AF 'LABEL'

Denne gang skal vi først registrere labelnavnet og den relative adresse, derefter skal vi kalde rutinen 'INGEN LABEL':

```
28000 S$=CHR$(PEEK(PT))+CHR$(PEEK(PT+1))
28010 SYM$(P)=S$:AB(P)=BC:P=P+1
28020 PT=PT+2
28030 GOSUB 26000
28040 RETURN
```

Og det var så det! Der er et par ting man skal passe på. For det første er der næsten ikke indbygget nogen fejlfangere. Det ville være en god idé at skrive en forbehandlingsrutine til at checke kodens syntaks, da et mellemrum for meget på

det forkerte sted eller et manglende kolon vil forvirre programmet fuldstændig. For det andet: glem ikke at tilføje linjen

```
10005 GOSUB 22000:REM ETABLER SYMBOLTABEL
```

På plussiden er der et par fine detaljer som 'bare var der'. Jeg kan ikke tage æren for dem, men de er der i hvert fald.

For det første kan man sætte to indledende mellemrum efter hvert linjenummer (undtagen når der står en label), sådan at man ser alting i kolonner på denne måde:

```
10      :A2 00      LDX #00
20L1    :A0 FF      LDY #FF
30      :BD 00 C0
40      :D0 L1      Løkke tilbage til L1 indtil 0 er nået
```

for BASIC fjerner dem nemlig igen. Det gør det vanskeliggere at komme til at lave fejl, når alting står pænt på rad og række. Så er de ekstra mellemrum ganske vist væk ved editering, men på det tidspunkt gør det ikke så meget.

For det andet behøver labels ikke begynde med 'L'. Reelt er begrænsningen at de skal begynde med et symbol med en ASCII-kode højere end 70, sådan at den udregnede 'byte' overskrider 255. Det betyder at ethvert bogstav fra 'G' og frem kan bruges. Du kan altså have flere labels end det er sandsynligt du vil få brug for, men hvis du vil have mere end 51 må du omdimensionere SYM\$ og AB. Der er for øvrigt heller ikke noget check for antallet af labels (over 51).

Start MINIASS med RUN 10000. 'GOTO' kan gå første gang man bruger det, men ikke de følgende, fordi symboltabellen ville blive dimensioneret igen. Skriv derpå startadressen for koden (ikke for dataområdet!) i decimal. Hvis du fx har 6 bytes data fra C000 til C005 er startadressen altså C006 = 49158.

Her er der et forslag til en ændring af MINIASS, som kan vise sig nyttig hvis du har flere separate rutiner, som du vil have assembleret og indlæst i forskellige områder af hukommelsen.

De kan se sådan ud:

10 :	}	første rutine
20 :		
30 :		
40 :		
50 *		
100 :	}	anden rutine
110 :		
120 :		
130 *		

og så videre.

Som det er nu kan MINIASS behandle dem alle sammen, hvis man assemblerer første rutine, sletter linje 10-50, starter MINIASS igen og så videre.

Men det ville være rart hvis man kunne indskrive første linjenummer af den rutine der skal assembleres, sådan at maskinen springer direkte til den rutine. Det er let at lave, for vi ved at de bytes som identificerer linjenumrene ligger på $LS + 2$ og $LS + 3$. Det eneste man behøver spekulere på er hvor symboltavlen skal dimensioneres!

Bemærk at symboltavlen i begge tilfælde initialiseres på ny for hver assemblering, og at labels derfor bliver behandlet som lokale labels for den enkelte rutine, så man kan godt genbruge labelnavne i flere rutiner efter hinanden uden at MINIASS bliver forvirret.

ET PROJEKT TIL

LOADER har flere funktioner som ikke findes i denne version af MINIASS: liste et maskinkodeprogram for at checke at det ligger rigtigt i hukommelsen, starte det, gemme det på kassette eller diskette, indlæse det igen fra kassette eller diskette. Man kan let pille de savnede rutiner ud af LOADER og kombinere dem med MINIASS for at få et virkeligt alsidigt hjælpemiddel. Den største mangel er noget der automatisk kan assemblere mnemonics som hexkoder. Hvis du har mod på at indtaste alle 151 mnemonics sammen med deres opkoder og antallet af bytes de bruger, samt tilføje nogle håndteringsrutiner, så kan du selv afhjælpe savnet. Det er en passende beskæftigelse til de lange vinteraftner.

Der er selvfølgelig meget mere at sige om maskinkode end det jeg har kunnet tage fat på her. Det her er bare begyndelsen. Hvis du vil nå meget længere får du brug for mere avanceret softwarehjælp – gør-det-selv-assemblering er ikke særlig velegnet til komplicerede programmer. Jeg vil afslutte med at samle:

22 · NOGLE LØSE ENDER

Der er nogle småting som jeg ikke har kunnet komme ind på endnu, men som jeg vil nævne inden jeg slutter. Først er der

LAGRING AF MASKINKODE OG BASIC I SAMME ARBEJDS GANG

Én mulighed er at indlægge de to rutiner ovenfor i BASIC-programmet. SAVE først BASIC-programmet, vælg så 'F' for at lagre maskinkoden i sin egen fil. Og omvendt: LOAD BASIC-programmet, tryk 'I' og indlæs filen i andet trin.

En anden mulighed er at POKE de systemvariabler der afgør hvor BASIC-program og variabler ligger, og således narre 64'eren til at rydde et område hvor maskinkoden kan ligge. Overfør maskinkoden til det område, og et enkelt SAVE vil nu lagre BASIC og maskinkode på én gang. Se *Programmer's Reference Guide* side 312 om adressering af systemvariabler (hvis du mener det så alvorligt med programmering har du allerede købt den!)

RELOKÉRBAR KODE

Al den snak om at flytte rundt på koden fører os ind på et andet emne. Hvis du undgår at bruge absolutte adresser i koden, kan den overføres til andre områder af hukommelsen uden problemer af nogen art. Det kendes under betegnelsen *relokérbar* (flyttelig) kode. Almindeligvis kan man fuldstændig undgå JMP, men JSR giver nogle problemer som kan gøre det nødvendigt at foretage lidt editering af koden før eller efter overførslen.

KERNAL

Man kan benytte alle rutinerne i 64'eren's ROM; alt hvad man skal vide er adressen på den pågældende rutine samt hvilken tilstand registrene skal have først.

Særlig nyttige er input-/output-rutinerne, som man får adgang til ved hjælp af et program der hedder KERNAL, som ligger i hukommelsen fra E000 til FFFF. En fuldstændig beskrivelse kan findes i *Programmer's Reference Guide*.

Her beskrives et par rutiner som du kan have særlig interesse i. Nogle KERNAL-rutiner kræver andre *forberedende rutiner* først. I så fald må de kaldes før hovedrutinen.

CHKIN: adresse FFC6

Åbner en kanal for input. Det logiske filnummer skal være i X-registret. Hvis du ikke skal bruge tastaturet som kommunikationsenhed må du bruge OPEN som forberedende rutine.

CHKOUT: adresse FFC9

Virker på samme måde som CHKIN, bare for output.

CHRIN: adresse FFCF

Henter en karakter fra inputkanalen og lægger den i akkumulatoren. Hvis input ikke kommer fra tastaturet kræves de forberedende rutiner OPEN og CHKIN. X-registret benyttes.

CHROUT: adresse FFD2

Som CHRIN, men for output. Forberedende rutiner er OPEN og CHKOUT, undtagen hvis output skal til billedskærmen.

CLALL: adresse FFE7

Lukker alle filer. Registrerne A og X benyttes.

CLOSE: adresse FFC3

Lukker en enkelt fil. Indlæs det logiske filnummer i akkumulatoren. X- og Y-registrene benyttes.

GETIN: adresse FFE4

Henter en karakter fra tastaturet (uden forberedende rutiner, men bemærk at det er tastaturbufferen der benyttes) eller anden input-enhed (forberedende rutiner CHKIN og OPEN). Indlæser karakteren i akkumulatoren.

OPEN: adresse FFC0

Åbner en logisk fil. De forberedende rutiner SETLFS og SETNAM skal benyttes.

PLOT: adresse FFF0

Viser cursorens position på skærmen. Kolonnennummeret skal være i X-registret, linjenummeret i Y-registret. Akkumulatoren benyttes.

SETLFS: adresse FFBA

Etablerer en logisk fil. Indlæs det logiske filnummer i akku-

mulatoren, enhedens nummer i X-registret og kommandoen i Y-registret (255 er default). Enhedernes numre er:

- 0 Tastatur
- 1 Kassetteoptager
- 2 Enhed med RS-232C-interface
- 3 Skærm
- 4 Printer med seriel bus
- 5 Diskettstation med seriel bus

SETNAM: adresse FFBD

Etablerer et filnavn. Navnets længde står i akkumulatoren, X- og Y-registret indeholder lille og store byte på den adresse i RAM hvor navnet begynder.

Lad os for eksempel sige vi vil PRINTe en karakter på skærmen på linje 7 kolonne 5. Lad os tage karakteren 'X' med ASCII-koden 58 hex. Brug først PLOT til at stille cursoren på linje 7 kolonne 5:

```
LDX #05      A2 05
LDY #07      A0 07
JSR PLOT     20 F0 FF
```

Brug nu CHROUT til at udskrive karakteren:

```
LDA #58      A9 58
JSR CHROUT   20 D2 FF
```

CHROUT-rutinen opdaterer automatisk cursorens position. Vi kan skrive 'HANS' på skærmen med:

```
LDA #48      A9 48      ASCII for 'H'
JSR CHROUT   20 D2 FF
LDA #41      A9 41      ASCII for 'A'
JSR CHROUT   20 D2 FF
LDA #4E      A9 4E      ASCII for 'N'
```


JSR CHROUT	20 D2 FF	
LDA #53	A9 53	ASCII for 'S'
JSR CHROUT	20 D2 FF	

Og husk så til slut:

RTS	60
-----	----

hvis du vil prøve det.

ASSEMBLERE

Til hjælp for editering og indskrivning af maskinkode fås der et antal assemblersystemer i handelen. De giver mulighed for at skrive med assemblermnemonics, bruge labels osv., og at omregne automatisk til hex.

En ting man skal være klar over er at de er *langsomme* at bruge, fordi de plejer at falde i flere dele, som skal indlæses enkeltvis i computeren: indlæs editeringsprogram, editér assemblerkode, læg assemblerkode i en fil, indlæs assembler, indlæs assemblerkode fra fil, læg hexkode i en anden fil, indlæs denne fil, udfør. Til små programmer er en assembler faktisk ikke meget værd.

Men til store programmer er en assembler derimod en nødvendighed. Commodore fremstiller én der hedder 64MON; der findes adskillige andre i handelen.

Man kan også skrive sin egen!

TILLÆG

1. OMREGNING MELLEM HEX OG DECIMAL

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
8	-128	-127	-126	-125	-124	-123	-122	-121	-120	-119	-118	-117	-116	-115	-114	-113	↑ tokomplement
9	-112	-111	-110	-109	-108	-107	-106	-105	-104	-103	-102	-101	-100	-99	-98	-97	
A	-96	-95	-94	-93	-92	-91	-90	-89	-88	-87	-86	-85	-84	-83	-82	-81	
B	-80	-79	-78	-77	-76	-75	-74	-73	-72	-71	-70	-69	-68	-67	-66	-65	
C	-64	-63	-62	-61	-60	-59	-58	-57	-56	-55	-54	-53	-52	-51	-50	-49	
D	-48	-47	-46	-45	-44	-43	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33	
E	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17	
F	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	↓ almindelig
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	

2. MNEMONICS

ADC	Add with carry Addition med mente
AND	Logical AND on each bit Logisk AND på hver bit
ASL	Arithmetic shift left Aritmetisk venstreforskydning
BBC	Branch if carry clear Forgrening ved slettet carry-flag
BCS	Branch if carry set Forgrening ved sat carry-flag
BEQ	Branch if result zero Forgrening ved nulresultat
BIT	Test bits from memory Test bit i hukommelsen
BMI	Branch if minus Forgrening ved negativt resultat
BNE	Branch if result non-zero Forgrening ved resultat forskelligt fra nul
BPL	Branch if plus Forgrening ved positivt resultat
BRK	Force break Afbrydelse af program
BVC	Branch if overflow clear Forgrening ved slettet overløbsflag
BVS	Branch if overflow set Forgrening ved sat overløbsflag
CLC	Clear carry-flag Slet carry-flag
CLD	Clear decimal mode flag Slet decimalflag
CLI	Clear interrupt disable bit Slet interruptflag
CLV	Clear overflow flag Slet overløbsflag
CMP	Compare accumulator with memory Sammenlign akkumulator med hukommelse

CPX	Compare index X with memory Sammenlign X-register med hukommelse
CPY	Compare index Y with memory Sammenlign Y-register med hukommelse
DEC	Decrement by one Formindsk med én
DEX	Decrement index X Formindsk X-register
DEY	Decrement index Y Formindsk Y-register
EOR	Exclusive OR Eksklusivt OR
INC	Increment by one Forhøj med én
INX	Increment index X Forhøj X-register
INY	Increment index Y Forhøj Y-register
JMP	Jump to new location Hop til anden adresse
JSR	Jump to subroutine Hop til underoutine
LDA	Load accumulator Indlæs i akkumulator
LDX	Load index X Indlæs i X-register
LDY	Load index Y Indlæs i Y-register
LSR	Logical shift right Logisk højreforskydning
NOP	No operation Ingen programudførelse
ORA	Logical OR Logisk OR

PHA	Push accumulator to stack Læg akkumulatorindhold på stak
PHP	Push processor status to stack Læg processorstatus på stak
PLA	Pull accumulator from stack Hent akkumulatorindhold fra stak
PLP	Pull processor status from stack Hent processorstatus fra stak
ROL	Rotate left one bit Venstrerotation med én bit
ROR	Rotate right one bit Højrerotation med én bit
RTI	Return from interrupt Retur fra interrupt
RTS	Return from subroutine Retur fra underoutine
SBC	Subtract with borrow Fratrækning med mente
SEC	Set carry flag Sæt carry-flag
SED	Set decimal mode Decimalindstilling
SEI	Set interrupt disable status Sæt interruptflag
STA	Store accumulator to memory Gem akkumulatorindhold i hukommelse
STX	Store index X Gem X-register
STY	Store index Y Gem Y-register
TAX	Transfer accumulator to index X Overfør akkumulatorindhold til X-register
TAY	Transfer accumulator to index Y Overfør akkumulatorindhold til Y-register
TSX	Transfer stack pointer to index X Overfør stakpointer til X-register
TXA	Transfer index X to accumulator Overfør X-registrets indhold til akkumulator

TXS	Transfer index X to stack register Overfør X-registrets indhold til stak
TYA	Transfer index Y to accumulator Overfør Y-registrets indhold til akkumulator

3. OVERSIGT OVER ADRESSERINGSMÅDER OG MNEMONIC-FORMATER

Følgende symboler bruges i dette tillæg:

MOP	Mnemonic for operation (fx STA)
11	Zeropageadresse (lille byte)
11 ss	Ikke-zeropage-adresse (lille byte, store byte)
oo	Offsetværdi: relativt tal mellem -128 og 127
tt	Talværdi
()	Indirekte adressering
,X	X-indeksering
,Y	Y-indeksering

Implicit og akkumulator-adressering

Enten behøves ingen adresse, eller også er akkumulator underforstået. Format:

MOP

Umiddelbar adressering

Numerisk værdi, ikke en adresse. Format:

MOP #tt

Absolut adressering (ikke-zeropage)

Benytter en to-bytes adressering. Format:

MOP 11 ss

Zeropage-adressering

Benytter en enkelt byte til at specificere en adresse på side 00. Format:

MOP 11

Præindekseret adressering (X-indeksering)

Adressering sker indirekte via en adresse indeholdt på side nul. Lille byte plus X angiver hvor adressen findes. Format:

MOP (11,X)

Postindekseret adressering (Y-indeksering)

Indirekte adressering med en adresse på side nul, hvortil Y-registrets indhold lægges. Format:

MOP (11),Y

Indeksring (fire muligheder)

Zeropage- og ikke-zeropage-adressering med X- eller Y-registret som indeks. Formater:

MOP 11,X

MOP 11,Y

MOP 11 ss,X

MOP 11 ss,Y

Indirekte adressering

To bytes angiver den adresse hvor den effektive adresse kan findes. Findes kun ved JMP. Format:

MOP (11 ss)

Relativ adressering

Et relativt offset lægges til programtællerens indhold. Bruges til forgrening. Format:

MOP 00

4. OPKODER FOR 6510

Tabellen viser opkoderne i hextal for alle mnemonics med alle tænkelige adresseringsmåder.

	Implicit	Umiddelbar	Absolut (ikke-zero-page)	Zero-page	Præindekseret med X	Postindekseret med Y	X-indekseret ikke-zero-page	X-indekseret zero-page	Y-indekseret ikke-zero-page	Y-indekseret zero-page	Indirekte	Relativ
ADC	-	69	6D	65	61	71	7D	75	79	-	-	-
AND	-	29	2D	25	21	31	3D	35	39	-	-	-
ASL	0A	-	0E	06	-	-	1E	16	-	-	-	-
BCC	-	-	-	-	-	-	-	-	-	-	-	90
BCS	-	-	-	-	-	-	-	-	-	-	-	B0
BEQ	-	-	-	-	-	-	-	-	-	-	-	F0
BIT	-	-	2C	24	-	-	-	-	-	-	-	-
BMI	-	-	-	-	-	-	-	-	-	-	-	30
BNE	-	-	-	-	-	-	-	-	-	-	-	D0
BPL	-	-	-	-	-	-	-	-	-	-	-	10
BRK	00	-	-	-	-	-	-	-	-	-	-	-
BVC	-	-	-	-	-	-	-	-	-	-	-	50
BVS	-	-	-	-	-	-	-	-	-	-	-	70
CLC	18	-	-	-	-	-	-	-	-	-	-	-
CLD	D8	-	-	-	-	-	-	-	-	-	-	-
CLI	58	-	-	-	-	-	-	-	-	-	-	-
CLV	B8	-	-	-	-	-	-	-	-	-	-	-
CMP	-	C9	CD	C5	C1	D1	DD	D5	D9	-	-	-
CPX	-	E0	EC	E4	-	-	-	-	-	-	-	-
CPY	-	C0	CC	C4	-	-	-	-	-	-	-	-
DEC	-	-	CE	C6	-	-	DE	D6	-	-	-	-
DEX	CA	-	-	-	-	-	-	-	-	-	-	-
DEY	88	-	-	-	-	-	-	-	-	-	-	-
EOR	-	49	4D	45	41	51	5D	55	59	-	-	-
INC	-	-	EE	E6	-	-	FE	F6	-	-	-	-

[illegible]

5. OPERATIONERNES INDVIRKNING PÅ FLAGENE

Tabellen viser alle operationer som har indvirkning på processorstatusregistret (flagene), samt hvordan flagene påvirkes. Operationer der ikke findes i listen har ingen indflydelse på flagene.

- * Sættes eller slettes afhængig af operationens resultat.
- 0 Slettes altid.
- 1 Sættes altid.
- 7 Sættes eller slettes som bit 7 i den pågældende byte.
- 6 Sættes eller slettes som bit 6 i den pågældende byte.

Operation	N	V	D	I	Z	C
ADC	*	*			*	*
AND	*				*	
ASL	*				*	*
BIT	7	6			*	
+ BRK						
CLC						0
CLD			0			
CLI				0		
CLV		0				
CMP	*				*	*
CPX	*				*	*
CPY	*				*	*
DEC	*				*	
DEX	*				*	
DEY	*				*	
EOR	*				*	
INC	*				*	
INX	*				*	
INY	*				*	
LDA	*				*	
LDX	*				*	
LDY	*				*	
LSR	0				*	*
ORA	*				*	
PLA	*				*	
PLP	*	*	*	*	*	*
ROL	*				*	*
ROR	*				*	*
RTI	*	*	*	*	*	*
SBC	*	*			*	*
SEC						1
SED			1			
SEI				1		
TAX	*				*	
TAY	*				*	
TSX	*				*	
TXA	*				*	
TYA	*				*	

+ BRK sætter også B-flaget

6. OPKODERNE I NUMMERORDEN TIL BRUG FOR DISASSEMBLERING

Koderne er angivet i hextal. Forkortelserne er de samme som i tillæg 3.

00 BRK	48 PHA	8E STX ll ss
01 ORA (ll, X)	49 EOR # tt	90 BCC dd
05 ORA ll	4A LSR	91 STA (ll), Y
06 ASL ll	4C JMP ll ss	94 STY ll, X
08 PHP	4D EOR ll ss	95 STA ll, X
09 ORA # tt	4E LSR ll ss	96 STX ll, Y
0A ASL	50 BVC oo	98 TYA
0D ORA ll ss	51 EOR (ll), Y	99 STA ll ss, Y
0E ASL ll ss	55 EOR ll, X	9A TXS
10 BPL oo	56 LSR ll, X	9D STA ll ss, X
11 ORA (ll), Y	58 CLI	A0 LDY # tt
15 ORA ll, X	59 EOR ll ss, Y	A1 LDA (ll, X)
16 ASL ll, X	5D EOR ll ss, X	A2 LDX # tt
18 CLC	5E LSR ll ss, X	A4 LDY ll
19 ORA ll ss, Y	60 RTS	A5 LDA ll
1D ORA ll ss, X	61 ADC (ll, X)	A6 LDX ll
1E ASL ll ss, X	65 ADC ll	A8 TAY
20 JSR ll ss	66 ROR ll	A9 LDA # tt
21 AND (ll, X)	68 PLA	AA TAX
24 BIT ll	69 ADC # tt	AC LDY ll ss
25 AND ll	6A ROR	AD LDA ll ss
26 ROL ll	6C JMP (ll ss)	AE LDX ll ss
28 PLP	6D ADC ll ss	B0 BCS oo
29 AND # tt	6E ROR ll ss	B1 LDA (ll), Y
2A ROL	70 BVS oo	B4 LDY ll, X
2C BIT ll ss	71 ADC (ll), Y	B5 LDA ll, X
2D AND ll ss	75 ADC ll, X	B6 LDX ll, Y
2E ROL ll ss	76 ROR ll, X	B8 CLV
30 BMI oo	78 SEI	B9 LDA ll ss, Y
31 AND (ll), Y	79 ADC ll ss, Y	BA TSX
35 AND ll, X	7D ADC ll ss, X	BC LDY ll ss, X
36 ROL ll, X	7E ROR ll ss, X	BD LDA ll ss, X
38 SEC	81 STA (ll, X)	BE LDX ll ss, Y
39 AND ll ss, Y	84 STY ll	C0 CPY # tt
3D AND ll ss, X	85 STA ll	C1 CMP (ll, X)
3E ROL ll ss, X	86 STX ll	C4 CPY ll
40 RTI	88 DEY	C5 CMP ll
41 EOR (ll, X)	8A TXA	C6 DEC ll
45 EOR ll	8C STY ll ss	C8 INY
46 LSR ll	8D STA ll ss	C9 CMP # tt

CA	DEX
CC	CPY ll ss
CD	CMP ll ss
CE	DEC ll ss
D0	BNE oo
D1	CMP (ll), Y
D5	CMP ll, X
D6	DEC ll, X
D8	CLD
D9	CMP ll ss, Y
DD	CMP ll ss, X
DE	DEC ll ss, X
E0	CPX # tt
E1	SBC (ll, X)
E4	CPX ll
E5	SBC ll
E6	INC ll
E8	INX
E9	SBC # tt
EA	NOP
EC	CPX ll ss
ED	SBC ll ss
EE	INC ll ss
F0	BEQ oo
F1	SBC (ll), Y
F5	SBC ll, X
F6	INC ll, X
F8	SED
F9	SBC ll ss, Y
FD	SBC ll ss, X
FE	INC ll ss, X

7. SPRITEREGISTRENE SKEMATISK

Adresse	Indhold								Funktion
D000	Kolonnenummer for sprite 0								Spritepositioner
D001	Linjenummer for sprite 0								
D002	Kolonnenummer for sprite 1								
D003	Linjenummer for sprite 1								
D004	Kolonnenummer for sprite 2								
D005	Linjenummer for sprite 2								
D006	Kolonnenummer for sprite 3								
D007	Linjenummer for sprite 3								
D008	Kolonnenummer for sprite 4								
D009	Linjenummer for sprite								
D00A	Kolonnenummer for sprite 5								
D00B	Linjenummer for sprite 5								
D00C	Kolonnenummer for sprite 6								
D00D	Linjenummer for sprite 6								
D00E	Kolonnenummer for sprite 7								
D00F	Linjenummer for sprite 7								
D010	Sp 7	Sp 6	Sp 5	Sp 4	Sp 3	Sp 2	Sp 1	Sp 0	Offsetflag
D015	Sp 7	Sp 6	Sp 5	Sp 4	Sp 3	Sp 2	Sp 1	Sp 0	Enable/disable
D017	Sp 7	Sp 6	Sp 5	Sp 4	Sp 3	Sp 2	Sp 1	Sp 0	Lodret udvidelse
D01D	Sp 7	Sp 6	Sp 5	Sp 4	Sp 3	Sp 2	Sp 1	Sp 0	Vandret udvidelse
D01E	Sp 7	Sp 6	Sp 5	Sp 4	Sp 3	Sp 2	Sp 1	Sp 0	Kollisionsflag
D027	Farvekode for sprite 0								Farver
D028	Farvekode for sprite 2								
D029	Farvekode for sprite 2								
D02A	Farvekode for sprite 3								
D02B	Farvekode for sprite 4								
D02C	Farvekode for sprite 5								
D02D	Farvekode for sprite 6								
D02E	Farvekode for sprite 7								
07F8	Datapointer for sprite 0								Pointere
07F9	Datapointer for sprite 1								
07FA	Datapointer for sprite 2								
07FB	Datapointer for sprite 3								
07FC	Datapointer for sprite 4								
07FD	Datapointer for sprite 5								
07FE	Datapointer for sprite 6								
07FF	Datapointer for sprite 7								

8. TASTATURSCANNINGSKODER

Skemaet viser hvilken værdi der findes i adresse 197 (00C5 hex) når en given tast er nedtrykket. Med PEEK(197) eller STA C5 (opkode 85 C5) kan værdien af den netop nedtrykkede tast aflæses uden om tastaturbufferen.

Tast	Kode	Hex	Tast	Kode	Hex	Tast	Kode	Hex
(ingen)	64	40		46	2E	T	22	16
*	49	31	A	10	0A	U	30	1E
+	40	28	B	28	1C	V	31	1F
.	47	2F	C	20	14	W	9	09
—	43	2B	D	18	12	X	23	17
.	44	2C	E	14	0E	Y	25	19
/	55	37	F	21	15	Z	12	0C
0	35	23	G	26	1A	RETURN	1	01
1	56	38	H	29	1D	CLR/HOME	51	33
2	59	3B	I	33	21	INST/DEL	0	00
3	8	08	J	34	22	CRSR ↓ ↑	7	07
4	11	0B	K	37	25	CRSR → ←	2	02
5	16	10	L	42	2A	←	57	39
6	19	13	M	36	24	f1	4	04
7	24	18	N	39	27	f3	5	05
8	27	1B	O	38	26	f5	6	06
9	32	20	P	41	29	f7	3	03
:	45	2D	Q	62	3E	£	48	30
;	50	32	R	17	11			
=	53	35	S	13	0D			

REGISTER

1-bytes kode 38
3-bytes kode 38
6510-processoren 30-31
absolut adressering 71
ADC 58
addition 35-40, 60, 62-64, 176-180
adresse 24
adresseringsmåder 70-74, 204-205
akkumulator 31, 105
akkumulatoroperationer 72
AND 144-145
ASL 67
assemblere 199
assemblersprog 57

Babbage, Charles 76
BASIC 184
BCC 83, 90
BCS 83, 90
BEQ 83, 90
bevægelse af sprites 159, 161
binærsystemet 14
binærtal 14-17, 20-23, 36
bit 16
bit-mapped grafik 174
bitlogik 144-145
BMI 83, 90
BNE 83, 90
Boole, George 144
boolesk algebra 144
BPL 83, 90
break-statusflag 81
brøktal 21
BVC 83, 90
BVS 83, 90
byte 24
bytelogik 146

carry-flag 58-59, 62, 79
centralenhed 30
CHKIN 196
CHKOUT 196
CHRIN 197
CHROUT 197
CLALL 197
CLC 58-59, 82
CLD 82
CLI 82
CLOSE 197
CLV 82
CMP 97
CPU 30
CPX 98
CPY 98

DEC 96
decimalflag 81
decimaltal 13, 16-19, 36, 200
dekrementering 96
DEX 98
DEY 98
disassemblering 131
division 64-66

EDIT 52
editor 182-193
enable/disable 153, 162
EOR 144-145

farve-RAM 129
farvehukommelse 12, 129-130
fil 53
FILE 53
flag 76, 80-81, 208-209
forgrening 83-92

forsinkelsesløkke 141-143
 fortegnslag 78
 fortegnstest 90-92

 gennemløbstæller 94
 GETIN 197

 heltal 21
 hexadecimalsystemet 15
 hexstal 15-19, 36, 200
 hi-res 169
 high byte 29
 hop 92-93
 hukommelse 24
 hukommelsesbanker 24-25
 højopløsningsgrafik 169-181
 højværdibyte 29

 implicit adressering 72
 INC 96
 indeks 101
 indekseret adressering 101
 indeksering 100
 indeksregistre 32, 97-98
 indirekte adressering 108, 111, 113,
 115
 inkrementering 96
 INPUT 54
 interruptflag 81
 INX 98
 INY 98

 JMP 92-93, 115
 JSR 117

 kassettebuffer 27, 42, 167
 KERNAL-rutiner 196-198
 kilobyte 24
 kollisionsflag 154

 labels 89, 187-191
 lager 24
 lagerceller 24
 lageropbygning 28
 lagring af maskinkode 53, 195

 lavværdibyte 29
 LDA 58
 LDX 98
 LDY 98
 lille byte 29, 61
 lo-res 169
 LOADER 43-47
 low byte 29
 LSR 69
 løkker 94-95, 97-99

 maskeret interrupt 81
 maskering 147
 memory 24
 mente 62-64, 66-68
 menteflag 79
 MINIASS 182
 mnemonics 57, 74-75, 201-204
 multiplikation 66-68, 136-137

 negative tal 21-23
 negativflag 78
 NOP 159
 nulflag 78

 offset 84, 101
 offsetflag 153, 161
 omvendt skærmfarve 134
 OPEN 197
 opkoder 39-40, 206-207, 210-211
 ORA 144-145
 ordlængde 20
 overflow 56, 80
 overløb 56
 overløbsflag 80

 P-register 33, 77
 page zero 27
 patch 140
 PC-register 33, 83, 118
 PEEK 25-26
 PHA 119
 PHP 120
 pixels 169
 PLA 119

PLOT 197
 plotning 172-180
 PLP 120
 pointer 42, 118, 153
 POKE 25-26
 postindekseret indirekte
 adressering 111
 print at 135
 PRINT 47
 processorstatusregister 33, 77
 programtæller 33
 præindekseret indirekte
 adressering 113

 RAM 24
 registre 31-32
 rekursiv programmering 117
 relokérbar kode 196
 ROL 66
 ROM 24
 ROR 65
 RTS 51, 118
 RUN 49

 sammenligning 96
 SBC 58
 SEC 59, 82
 SED 82
 SEI 82
 selvændrende program 100, 106-107
 SETLFS 197
 SETNAM 198
 side nul 27
 skærmfarve 129
 skærmhukommelse 12, 126-128,
 132-133
 SP-register 118
 spritedata 167
 spritedatapointere 154-155

 spritekollision 167
 spritekonstruktion 149, 151, 157
 spritekoordinater 155
 spritepositioner 153
 spriteprioritet 164
 spriteregistre 153, 156, 212
 sprites 149-168
 STA 57
 stak 33, 117-125
 stakpil 33, 118
 standardområdet
 (49152-53247) 41-42
 store byte 29, 61
 STX 98
 STY 98
 subtraktion 58-59
 SYS 49

 tastaturkontrol 138, 162
 tastaturskanningskoder 213
 TAX 105
 TAY 105
 titalssystemet 13
 tokomplement 21-23
 totalssystemet 14
 TXA 105
 TYA 105

 udvidelse af sprites 154, 161
 umiddelbar adressering 70
 underrutiner 117-125

 X-register 98, 101-103, 105, 113

 Y-register 98, 101, 105, 112

 zeroflag 78
 zeropage 71, 102, 110

Maskinkode med Commodore 64 er en let-tilgængelig og grundig introduktion til programmering af 6502 og 6510 i maskinkode og assembler, og den forudsætter ikke nogen kundskaber hos læseren ud over et beskedent kendskab til BASIC.

I bogen benyttes et BASIC-program (en loader) for at gøre det nemmere at opstille og indlæse maskinkodeprogrammerne.

Bogen indeholder en lang række enkle maskinkodeprogrammer, der er lette at forstå og som læseren kan benytte som byggesten i mere komplicerede programmer.

I de afsluttende kapitler lærer læseren at udnytte 64erens særlige egenskaber, herunder at styre *sprites* og *højopløsningsgrafik*, samt kontrollere tastaturet ved hjælp af maskinkode.

Når du har læst bogen vil du føle dig hjemme i 6502 og 6510 maskinkode og have erhvervet et grundlæggende kendskab til en række af de programmeringsmæssige hemmeligheder ved din Commodore 64.

BORGEN



**This was brought to you
from the archives of**

<http://retro-commodore.eu>